TEK PROGRAMMERS REFERENCE

Part No. 061-3209-00
Product Group 07

4400 SERIES ASSEMBLY LANGUAGE

Tektronix
COMMITTED TO EXCELLENCE

TEK PROGRAMMERS REFERENCE

Part No. 061-3209-00
Product Group 07

4400 SERIES ASSEMBLY LANGUAGE

Please Check for CHANGE INFORMATION at the Rear of This Manual

First Printing MAR 1986

Tektronix®
COMMITTED TO EXCELLENCE

WARRANTY FOR SOFTWARE PRODUCTS

Tektronix warrants that this software product will conform to the specifications set forth herein, when used properly in the specified operating environment, for a period of three (3) months from the date of shipment, or if the program is installed by Tektronix, for a period of three (3) months from the date of installation. If this software product does not conform as warranted, Tektronix will provide the remedial services specified below. Tektronix does not warrant that the functions contained in this software product will meet Customer's requirements or that operation of this software product will be uninterrupted or error-free or that all errors will be corrected.

In order to obtain service under this warranty, Customer must notifiy Tektronix of the defect before the expiration of the warranty period and make suitable arrangements for such service in accordance with the instructions received from Tektronix. If Tektronix is unable, within a reasonable time after receipt of such notice, to provide the remedial services specified below, Customer may terminate the license for the software product and return this software product and any associated materials to Tektronix for credit or refund.

This warranty shall not apply to any software product that has been modified or altered by Customer. Tektronix shall not be obligated to furnish service under this warranty with respect to any software product a) that is used in an operating environment other than that specified or in a manner inconsistent with the Users Manual and documentation or b) when the software product has been integrated with other software if the result of such integration increases the time of difficulty of analyzing or servicing the software product or the problems ascribed to the software product.

TEKTRONIX DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. TEKTRONIX' RESPONSIBILITY TO PROVIDE REMEDIAL SERVICE WHEN SPECIFIED, REPLACE DEFECTIVE MEDIA OR REFUND CUSTOMER'S PAYMENT IS THE SOLE AND EXCLUSIVE REMEDY PROVIDED TO CUSTOMER FOR BREACH OF THIS WARRANTY. TEKTRONIX WILL NOT BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES IRRESPECTIVE OF WHETHER TEKTRONIX HAS ADVANCE NOTICE OF THE POSSIBLITY OF SUCH DAMAGES.

PLEASE FORWARD ALL MAIL TO:

Artificial Intelligence Machines
Tektronix, Inc.
P.O. Box 1000 M.S. 60-405
Wilsonville, Oregon 97070
Attention: AIM Documentation

MANUAL REVISION STATUS

PRODUCT: 4400 SERIES ASSEMBLY LANGUAGE PROGRAMMERS REFERENCE

This manual supports the following versions of this product: **4404 Version 1.5, 4405 Version 1.1, and 4406 Version 1.1 .**

REV DATE	DESCRIPTION
MAR 1986	Original Issue

Table of Contents

Figures

Tables

Section 1
INTRODUCTION

ABOUT THIS MANUAL

This manual is the primary programmer´s reference to the 4400 assembly language. This manual contains a guide to assembly language programming and the system calls that you can use with the assembler. The *4400 Users Manual* contains a complete list of the other manuals available for the 4400 series.

This manual has the following sections:

Introduction	About this manual.
Programmers Guide	A general introduction to assembly language programming.
The Assembler	A description of the assembler and linking loader.
System Calls	A description of the system calls available to the assembler.

The 4400 series assembler is similar on all 4400 series products. When assembling mnemonics, the assembler generates code that fits the processor of the machine it is running on. To assemble code for a different processor, set the swiches on the command line appropriately. Information for setting the switches is found in section 3, *The Assembler*, of this manual. If you write a program that uses MC68020 mnemonics, by proberly setting the command line switches it will assemble and run on a MC68010 or MC68000 processor. This simplifies writing and assembling programs for use with other microprocessors.

WHERE TO FIND INFORMATION

You have several important sources of information on the 4400:

- This manual, the *4400 Series Assembly Language Programmers Reference* manual, contains the details of the assembler and linking loader.
- The *4400 Series Operating System Reference* manual contains the syntax and details of commands and utilities. This manual also contains details about a text editor and a remote terminal emulator.
- The *4400 Series C Language Reference* manual contains detail about the "C" programming language.
- The *4400 Users* manual contains basic information on system installation, startup, installing software, and the other "how to put commands together" discussions. See the index of the *User´s* manual to find how to perform particular tasks.
- The on-line "help" utility contains a brief description of the syntax of user commands.
- The *Introduction to Smalltalk-80(tm)* manual contains details and a short tutorial on the Smalltalk-80 programming language.
- The reference manuals for the optional languages for the 4400 product family are also available.

MANUAL SYNTAX CONVENTIONS

Throughout this manual, the *4400 User's* manual, and in the on-line help files, the following syntax conventions apply:

1. Words standing alone on the command line are *keywords*. They are the words recognized by the system and should be typed exactly as shown.
2. Words enclosed by angle brackets (< and >) enclose descriptions that are replaced with a specific argument. If an expression is enclosed *only* in angle brackets, it is an essential part of the command line. For example, in the line:

   ```
   adduser <user_name>
   ```

   you must specify the name of the user in place of the expression <user_name>.
3. Words or expressions surrounded by square brackets ([and]) are optional. You may omit these words or expressions if you wish.
4. If the word *list* appears as part of a term, that term consists of one or more elements of the type described in the term, separated by spaces. For example:

   ```
   <file_name_list>
   ```

   consists of a series (one or more) of file names separated by spaces.

Section 2

PROGRAMMER´S GUIDE

INTRODUCTION

This section, the 4400 Programmer´s Guide, provides a general introduction to MC68000/68010/68020 assembly language programming on the 4400 product family. This section includes a sample 4400 utility program that you can type in and execute.

For information on the MC68000/68010/68020 assembler, see Section 3, *The Assembler and Linking Loader*. For information on system calls, see Section 4, *System Calls*. System programming in C is described in the manual, *THE 4400 C COMPILER*, while programming in other languages is described in the reference manuals for those languages.

SYSTEM CALLS OVERVIEW

The following paragraphs give an overview of assembly language programs of the 4400 family: how they run, how they perform system function calls, how they handle errors, and what the task environment is like.

How 4400 Programs Run

Most programs or utilities are run by typing the name of such a program in response to a prompt from the shell. The shell assumes the typed name is a file containing an executable binary program. (There are exceptions, such as command text files, but we will ignore those for now). This binary program is loaded into memory and executed. If desired, this program can obtain parameters from the command line. When it is finished, the program terminates, passing control back to the shell.

Every program that runs on the system is a task. Many tasks may be active at once, but in reality only one task is running at any given instant. The system switches from task to task so rapidly that the appearance is that all of the tasks are executing concurrently. If you were to freeze the system at some point in time, you would see a single task or program in the cpu´s address space. A task may not have all of RAM assigned to it, but it would have the entire address space available. Other tasks may be resident in other memory, but that memory is not mapped into the task´s address space. When the task terminates, its allocated memory is returned to the system, and control is passed to the parent task (the task which created or initiated the terminating task).

This section discusses how to write a program which the shell can load and execute, how this program can communicate with the user, system, other tasks, etc, and how to terminate the program´s execution.

INTRODUCTION TO SYSTEM CALLS

When a user's program communicates with the user, a disk file, another task, or anything else in the system, it uses *calls* to the operating system. The operating system is essentially another task, always available, that has built in routines to perform a variety of system oriented functions. These functions include reading files, writing files, seeking to file locations, setting permissions, creating pipes, reporting id's, creating tasks, terminating tasks, mounting devices, reporting the time, and so on.

A user program executes functions by making a call to the system with a proper function code and input parameters. The technique of making the *call* in the assembler code is the *sys* instruction recognized by the assembler. In addition to the *sys* calls, which implement generic Operating System instructions, other calls made with the *trap* instruction allow access to the 4400 series specific hardware, such as the display or floating-point processor.

The *sys* Instruction

The assembler has a built-in instruction to make system calls. It is the *sys* instruction and has the following format:

```
        sys <function>,[<parameter1>,...<parameter4>]
```

The only required portion of the operand is the <function>, which is a numeric code for the desired function. The parameters required depend on the particular function. There may be no parameters or as many as four. The function code is a 16-bit value; while parameters are always 32-bit values. Many system functions also require certain values or parameters to be in one or more of the processor's cpu-registers before executing a *sys* instruction. When some parameters are required in registers, it is the programmer's responsibility to see that the proper values are loaded before calling on the system.

When the *sys* instruction has completed execution, control generally passes to the next instruction in the program. In some cases, the system function returns one or more values to the calling program by placing the values in selected cpu registers. In some cases the returned value(s) are placed at a location specified as one of the input parameters.

Section 4, *System Calls*, describes the operating system functions. Along with the description, the necessary parameters and returned values are specified. For example, look at the *read* system call in that section. Under the USAGE heading you will see the following:

```
<file descriptor in D0>
sys read,buffer,count
<bytes read in D0>
```

This shows that before executing the read function call, you must ensure that the desired *file descriptor* must be loaded into the processor's D0 register. In addition to the read function code itself, you must supply a buffer address (32-bit address of a buffer to read into) and a count (32-bit count of how many characters to read). After executing the read function, the actual number of bytes read is returned in the processor's D0 register.

All user-accessible processor registers except for the D0, A0, and CCR registers are left intact across system calls. The contents of the D0, A0, and CCR registers upon return from a system call vary depending on the particular call.

The actual system function code numbers are defined in the *sysdef* file located in the */lib* directory. This file is provided on disk so that you can include those definitions in your program by including the *sysdef* file in your source via a *lib sysdef* instruction.

Briefly, the *sys* function works by generating a software interrupt. When this interrupt occurs, the handling routine maps the calling task out of the cpu's address space and maps the operating system code in. This system code then performs the requested function. It obtains the function number and parameters from the code directly following the software interrupt itself. When the system function has completed, the operating system is mapped out, and the task is mapped back in, to continue with its instructions.

System Call Example

Let's try a sample program that includes a system function call. The sample program has four fields: Label, Opcode, Operand and Comment. In section 3, *The Assembler*, of this manual there is a description of the source statement fields (columns) in a program. The four source statement fields are summarized as:

Label	Contains a symbolic label or name that can be called upon throughout the source program.
Opcode	Contains the opcode (mnemonic) or pseudo-op.
Operand	Provides data ar address information required by the opcode.
Comment	Contains comments on each line of code.

The simplest program is one that does nothing at all: as soon as it is initiated, it immediately terminates. Thus, the only system function we will need to call is the *term* function. The description of *term* in Section 4, *System Calls*, shows that there are no parameters required on the *sys* instruction itself (besides the function code), but that you must put a status value in the D0 register before performing the call. If there are no errors this status should be zero. Thus you can write an extremely simple program that looks like the following:

```
Label   Opcode   Operand   Comment

        lib      sysdef
        text
start   move.l   #0,d0     Put status in D0
        sys      term      Terminate task
        end      start
```

The first line includes the definitions of all system function codes so that we can specify the term function as a symbol (*term*) and not have to type in the particular number for that function. The third line puts the status in D0, as required by the *term* function, and line 4 terminates the program. In the case of the *term* function, control is not returned to the calling program after execution of the call. Of course, that is the reason for the function; it terminates the current task (the task which made the call) and returns control to that task's parent. Notice that the program's end statement includes the symbol *start*. This tells the assembler the beginning location for execution and also induces the assembler to make the resulting code executable by setting the permission bits.

Let´s assume you call the source file *nothing.asm* and assemble it with the following commands:

```
        ++ asm nothing.asm +s +o=nothing.r
        ++ load nothing.r +o=nothing
```

The result would be a binary file that when executed by the command:

```
        ++ nothing
```

would load, run, and immediately return to the shell. This is, of course, a meaningless example, but it does show the rudimentary steps in writing, assembling, and executing a 4400 assembly language program.

Indirect System Calls

In order to use the *sys* instruction directly, you must define all the parameters at assembly time. When parameters are not known at assembly time (because they will be determined or changed during the execution of the program), you must use *indirect system calls.* There are two types of indirect system calls — *ind* and *indx* — and they are themselves system functions called with the normal sys instruction. They permit the programmer to tell the system that the parameters do not actually follow the software interrupt, but instead are placed at some other specified location in memory. This memory location, specified by the programmer, can be in an area of memory containing data and not program code.

The first of these indirect system call functions is called *ind.* Its format is:

```
        sys ind,label
```

The *label* is the address of the memory locations that contains the actual desired function code and parameters. Thus, when this function is executed, the system goes to location *label* and picks up the desired function code and any necessary parameters. The system executes that function and returns control to the statement following the *sys ind,label* instruction.

To illustrate, let´s assume a program that needs to read from a file, but does not know how many characters to read until it is executing. Somewhere in the first part of the executing program, the number of characters to be read is determined and stored in a label called *rcount.* The indirect function call is used:

```
        ...
        ...
        move.l  rcount,iread+6   Put count to read
        move.l  fd,d0            Put file descriptor
        sys     ind,iread        Do indirect read call
        ...
        ...
iread   dc.w    read             READ function code
        dc.l    buffer           Read buffer location
        dc.l    0                Read count (unknown)
buffer  ds.b    $4000            Space for read buffer
```

(At this point we're not concerned with details of how the read really works or what the file descriptor is, we simply want to show how the indirect system call is made.)

The second form of indirect system call is the *indx* function, and is very similar to the *ind* function. The difference is that the call to *ind* includes a parameter (*label*) that points to the parameters in memory; with the *indx* function the pointer to the parameters in memory is in the A0 register. To see how this works, we can modify the above sample by changing the instruction *sys ind,iread* to:

```
    lea    iread,a0    Get address of parameter
    sys    indx        Do indirect read call
```

An obvious use of indx is to push the parameters onto the system stack and point A0 to it, thereby eliminating the need for the parameter buffer in memory. For example:

```
        ...
        ...
        move.l   rcount,-(a7)    Set count to read
        move.l   #buffer,-(a7)   Set buffer address
        move.w   #read,-(a7)     Set read function code
        move.l   fd,d0           Put file descriptor
        move.l   a7,a0           Point to parameters on stack
        sys      indx            Do indirect read call
        lea      10(a7),a7       Clean parameters off
        stack
        ...
        ...
buffer  ds.b     $4000           Space for read buffer
        ...
        ...
```

Note the importance of the order in which the parameters are pushed onto the stack. Also note the *lea 10(a7),a7* instruction following the function call. It removes the parameters which were pushed onto the stack so that the stack is where it was before the system call section.

HARDWARE ACCESS TRAPS

The 4400 series Operating System supports direct user program access to hardware facilities such as the bitmap display, mouse, keyboard and interval timer. The *Trap #13* instruction provides this access. Set register *d0* to a value that indentifies the specific function to be performed when a *Trap #13* instruction is executed.

Symbolic names for these function codes are defined in the file */lib/sysdisplay*. Section 5, *Display Access Functions* and section 6, *Keyboard and Mouse Functions*, describes the details of these functions.

FLOATING POINT TRAPS

The 4404 uses a National Semiconductor 32081 Floating Point co-processor. This is interfaced to the MC68010 as an I/O device that is accessably only when the processor is operating in supervisor mode. The 4400 operating system provides routines that allow programs to perform floating point calculations using the NS32081. Invoke these routines using the *Trap #12* instruction. When a *Trap #12* instruction is executed, specify the floating point function code in register *d0*.

These function codes are defined in the file */lib/sysfloat*. Section 7, *Floating Point Functions*, describes the details of the support.

The MC68020 based members of the 4400 series do not use the NS32081 floating point processor. Instead, they use the Motorola MC68881 floating point processor. This processor implements a superset of functions available with the NS32081. Additionally, the MC68881 floating point processor are optionally accessable to the programmmer without the use of operating system trap routines.

To maintain compatability with the 4404 programs, MC68020 members of the 4404 series family also implement the floating point trap routines as an alternate way to access the MC68881.

SYSTEM ERRORS

Upon completion, system calls return to the calling program with an error flag. This flag is the carry bit in the condition code register. If the bit is zero on return, it implies that no error occurred. If the bit is set (a one), then an error has occurred and the D0 register contains an error number. The assembler supports two special mnemonics for testing the error status on return from a system call: *bes* for *branch if error set* and *bec* for *branch if error cleared.* These are equivalent to the standard mnemonics *bcs* and *bcc.*

Section 4, *System Calls,* contains a list of the error numbers and their meanings. There is also a file of equates called */lib/syserrors* which assign standard labels to the error numbers. These can be used in a program by simply including the file with a *lib syserrors* instruction. Note that the operating system does not report errors directly to the user. Error numbers are returned from system calls and it is entirely up to the user's program to report such errors or handle them as required by the specific application.

The Task Environment

A *task* is a single program which has complete use of the cpu's directly-accessible address space. It can call on functions in the operating system, but is essentially a single, stand-alone program. Each time a program is run, a new task is generated and the program becomes that task. Whenever that executing task performs some I/O or system call that requires it to wait, the task is mapped out so that another waiting active task may be mapped in and executed. If the executing task does not perform any type of system call which would cause it to be mapped out, it will eventually run into a time-slice interrupt which forces the task out so that other tasks can get some execution time.

In this manner, multiple tasks can be run at what seems like the same time. To assist in keeping track of all the active tasks, the operating system assigns a unique *task id* number to each task. This is a 15 bit unsigned value that can be used to uniquely identify a particular task. The *gtid* system call allows a task or program to obtain this task id if desired.

Address Space

The addresses which can be generated by a program make up what is known as the logical address space. Under hardware memory management, these logical addresses are not presented directly to the system memory. Instead, they are routed through the hardware memory manager, which translates the logical addresses into physical addresses. Memory management allows programs which reside at a particular logical address to actually load into system memory at a different physical address. The total range of physical addresses makes up the physical address space.

Although it would be possible to pass the addresses generated by the program directly to the system memory, the use of a hardware memory manager provides several benefits. First, and perhaps foremost, it prevents one task from reading from, or writing to, the memory allocated to another task. In addition, it allows multiple tasks to reside in physical memory without the need for each task to reside in a different area in the logical address space. Thus, all programs can be written to execute at the same fixed logical address. No matter where those programs are loaded into physical memory when they are executed, the memory management unit converts the logical addresses the program uses to the proper physical addresses.

The 4400's logical address space is divided into four sections: text, data, stack and shared resources. The program itself resides in the text section. This section cannot be written to during execution of the program. The data section contains any data the program uses. It can be both read from and written to during execution. The system stack is located in the stack section. The shared resource section is an area where resources shared by tasks, such as the display bitmap, may be accessed. These resources are made addressable by using the *phys* system call.

The memory management unit allocates a certain amount of memory to each section when the task is initiated. The amount of memory assigned to each section is determined by the size of the task and its needs. It is also possible, as we shall see later, for a task to add more memory to the data or stack section during execution.

The address space of a task is futher divided into smaller units called pages. A page is the smallest unit of memory controllable by the memory management hardware. Using the *mmeman* system call, individual pages may be protected, added, or deleted, from a tasks address space.

The size of a page and the maximum size of a tasks address space varies amoung the members of the 4400 series family. Refer to the appendices of the *Operating System Reference* for more information about page and maximum task size.

Arguments and Environments

It is often desirable to pass arguments or parameters to a program when you begin its execution. The *exec* and *execе* system calls provide this ability. *Exec* and *execе* are the calls that are used to begin execution of a program or binary file.

Arguments are passed to a program by leaving them on the system stack. When initiating a program, the system stack pointer (A7) is left pointing at some unknown location in the stack page. Any arguments passed to the program are found in a special format just above where the stack pointer points. The environment variables are also found in this area.

The arguments themselves are simply strings of characters which the program must know how to use. In order to easily find these strings, the system provides a list of pointers to the beginning of the strings. In addition, the system provides a count of how many arguments have been passed.

The pointers to the environment variables are found in memory, directly above the pointers to the arguments. Since there is no count of the pointers to the environment variables, they are terminated by a null string. Refer to Figure 2-1 for the relationship of the pointers to the strings.

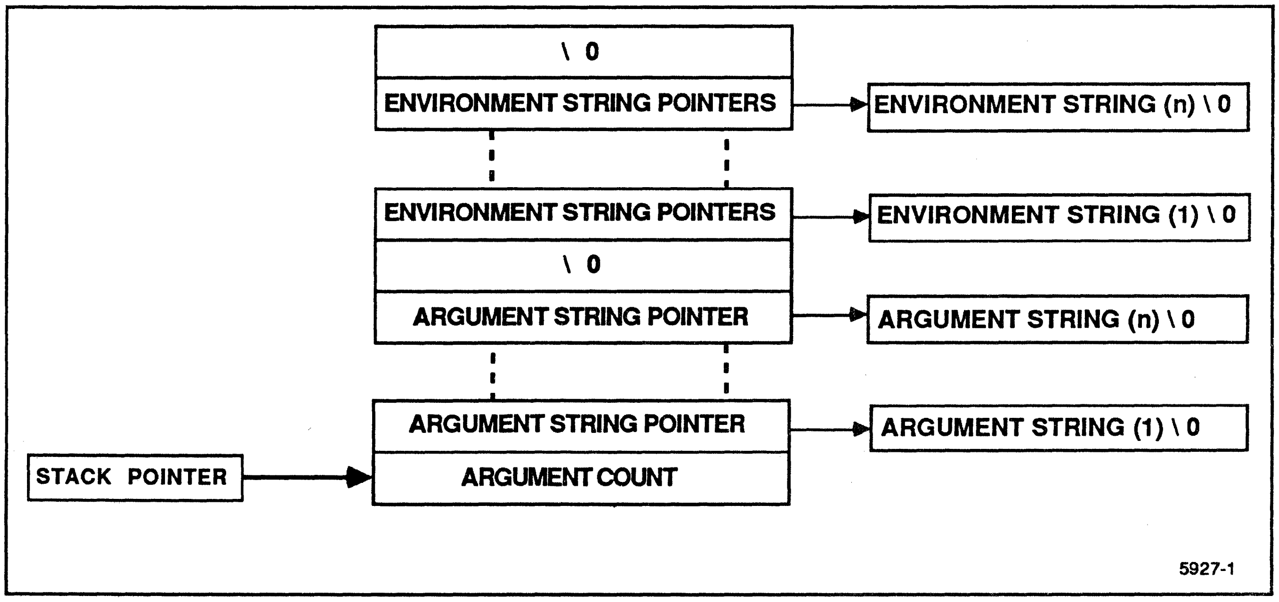

Figure 2-1. Relationship of Passed Pointers.

This argument information is laid out as follows:

1. The stack pointer is pointing to the argument count. It is a 4 byte value and should always be greater than zero.
2. Just above the argument count (higher addresses in memory) is the list of pointers to the argument strings. These pointers are 32 bit addresses of the actual strings.
3. At the end of the pointer list are four bytes of zero to signify the end of the list. (A null pointer.)
4. The list of pointers to the environmental variables is next. these pointers are 32 bit addresses to the actual strings.
5. At the end of the pointer list are four bytes of zero to signify the end of the list. (A null pointer.)
6. The actual argument strings begin above the zero bytes. Each argument string is the string of characters that make up the argument followed by a zero byte.
7. The environment strings are next. Each string is the string of characters that make up the environment variables followed by a zero byte.
8. An additional null string after after the terminal null of the last environmental variable string terminates the null string.

In general, the shell initiates the programs or utilities that a system programmer writes. Specifically, they are started when the user types the name of that program in response to the shell´s prompt. The shell starts the program by performing an *exec* system call. The arguments that the shell sets up for the exec (which are those passed to the program) are the arguments that are typed on the shell command line after the program name. By convention, the shell sets argument 0 to be the command or program name itself. The arguments after the program name are then numbered sequentially beginning with one.

The shell performs pattern-matching before passing the arguments to the command. For example, consider the command:

```
++list file*
```

The shell does not pass *file** as an argument to list, but rather searches the directory for all filenames that match and passes them all as individual arguments. Thus, the list program would see four arguments:

```
argument 0 -> list
argument 1 -> file1
argument 2 -> file2
argument 3 -> filename
```

(Recall that argument number zero is always the name of the program or command being executed.)

INITIATING AND TERMINATING TASKS

In a multi-tasking environment, one task can spawn or start a new task. There must, of course, also be means for terminating tasks and for the parent of a terminating task to be informed of that termination. The following discussion covers these techniques.

Terminating a Task

Tasks or programs are terminated with the *term* system call. When this function is executed, the task is halted and its memory is relinquished to the system. Before calling the *term* function, the programmer is required to place an error status value in the D0 register. When the task terminates, this value is passed back to the task's parent. If there is no error on termination, this error status should be zero to indicate a clean termination. If the task terminates due to a system error such as an I/O error, the error value returned by that system call should be used as the error status for the term function. If the task terminates due to an error defined by the program (for example, the program expects an argument but none was supplied), the recommended value to return is $000000FF. By convention the parent task would recognize this as a user-defined error. The parent would know some error had occurred that caused the program to terminate, but would not be able to determine the exact nature of the error. A user-defined error should not return a termination status of greater than $000000FF.

The *wait* System Call

The *wait* system function is issued by a task when it wants to wait for one of the child tasks it has spawned to terminate. It is through the *wait* command that the parent task receives the termination status from its child. *Wait* has the following syntax:

```
        sys wait
```

When the system call *wait* returns, the termination status is in the A0 register and the terminated task's id is in the D0 register.

If there are no child tasks when a *wait* call is issued, an error is returned. If a child task is still running when the parent issues a *wait*, the parent is put to sleep until the child task terminats. If a child task terminates before its parent has issued a *wait*, the system will save the child's task id and termination status until the parent does issue a *wait*. If several child tasks have been spawned, the parent must issue a *wait* call for each one individually.

The termination status is a two-byte value that is returned in the lower half of the A0 register. The lower byte (bits 0-7 of A0) is the low-order byte of the status value passed by the *term* system call. If this byte is non-zero, some sort of error condition caused termination. Under normal termination conditions, the higher byte of the termination status (bits 8-15 of A0) is zero. If non-zero, the task was terminated by some system interrupt, and the least significant seven bits of this byte contain the interrupt number. If the most significant bit of this byte is set, a core dump was produced as a result of the termination. (Interrupt numbers and core dumps are described later.)

The *exec* System Call

At times, a user-written program may wish to load and execute a program by itself without using the shell. The tool used to load and execute another program or binary file is the *exec* system function. That is the function which the shell uses when it loads and executes a program. (Remember the shell itself is just another program.)

The program that makes the *exec* call and the new program (a binary file) have the same task id number. If the *exec* is successful (i.e. no errors such as the file not existing), there is no return to the calling program. The calling program is thrown away, making it impossible to return. If, however, there is an error in attempting to perform the *exec* function, the system does not load the new program and returns an error status to the calling program, which is still intact. Thus a properly written program follows any *sys exec* call with error handling code.

If the environment variables are to be passed to the new program, use the *exece* system call.

The *exec* call requires two arguments: a pointer to the name of the file to be executed and a pointer to a list of arguments to be supplied to the new program. *Exec´s* format is:

```
        sys exec,fname,arglist
```

The *fname* is the pointer to the filename. This filename is a string of appropriate characters located somewhere in memory and terminated by a zero byte. The *arglist* is the pointer to a list of argument pointers. In other words, *arglist* is an address that points to an address that begins a list of pointers to arguments. This list of pointers is consecutive 4-byte addresses or pointers to the actual argument strings. The list is terminated by four bytes of zero (which could be considered a pointer to zero). Each pointer in the list is the address of the actual argument string that is terminated by a zero byte. When the *exec* function is complete, the new program will have these arguments available in the exact format previously described.

Let´s try an example of the use of *exec*. As you know the *dir* command can be run by typing the name and possible arguments on the shell command line. The shell actually starts execution of *dir* by performing an *exec*. As an exercise, let´s write our own program that executes the *dir* command automatically, always providing an argument of *+ba*. This provides a long directory with file sizes specified in bytes and includes all files. We will not specify any specific directory, so our command will always perform the directory command on the current directory. The filename to *exec* should be */bin/dir*, and there are two arguments, *dir* and *+ba*. We supply *dir* as argument zero because by convention argument number 0 is the command name. Our program looks like this:

```
        lib     sysdef

        text
start   sys     exec,filen,args

* This point is reached only if the exec fails. There
* would normally be error handling code here, but to keep
* things simple, we will just terminate if an error.
* Note the D0 register already has the error from exec.
```

```
        sys     term
* strings and data

filen   fcc     '/bin/dir',0
arg0    fcc     'dir',0
arg1    fcc     '+ba',0
args    dc.l    arg0,arg1,0
        end     start
```

If we called this utility *ldir*, after assembling we could execute it by typing *ldir* as a command to the shell. Our program would be loaded and executed by the shell, and it would in turn load and execute the *dir* command with a *+ba* option. Thus typing *ldir* would produce the same results as typing *dir +ba.*

The *fork* and *vfork* System Calls

The *fork* and *vfork* system calls are used to spawn a new task, and are the only way to create new tasks. The *fork* system call creates a new task which is almost identical to the old task (the old task still exists). The *vfork* system call is more efficient by creating a new task with the same memory and stack allocation, same code in the memory space, same open files, pointers, etc. (See section 4, *System Calls*, for a more complete description.)

Thus, immediately after a fork, there are essentially two identical tasks or programs running on the system. Usually you want the new task to do something different, so in most cases the new task immediately performs an *exec* call to load some program and execute it. This is the technique used by the shell to start background jobs. When the shell sees a command ending with an ampersand (&), instead of directly doing an *exec* it does a fork to create a second shell. Now the newly created shell will do an *exec* of the desired command, while the old shell is still around to accept further commands.

The syntax of either fork command is simply:

```
        sys fork
```

or

```
        sys vfork
```

The subtle part of the *fork* call is in how the two almost-identical tasks know which is which. If the two tasks have the same code, how can the new one do an exec while the old one does not? The answer is in the return from a fork call. After the fork operation, execution resumes in each of the two programs. The difference is in *where* that execution resumes. In the new task, execution resumes in the instruction immediately following the fork system call. The old task resumes execution at a point two bytes past the system call. In this manner, the same program can be run in two tasks via a fork and yet do different things after the fork. Since the new task resumes directly after the fork call and the old task resumes two bytes after the fork call, it is obvious that the first instruction in the new task must be a short branch instruction (which requires only two bytes). Note that the new task's id is made available to the old task by supplying the id in the D0 register upon return from the fork. If an error occurs when attempting a fork, the new task is not created, and an error status is returned to the old task (still two bytes past the fork system call).

The *vfork* system call is used when the new task will immediately perform an *exec* call. *Vfork* avoids making a complete copy of the parent tasks address space since the parent task is completely discarded by the *exec* call.

The following section of code helps illustrate the fork:

```
        ...
        ...
        sys       fork              spawn new task

* new task begins execution here

        bra.s     newtsk            branch to code for new task

* old task resumes execution here

        bes       frkerr            check and branch if error
        move.l    d0,d1             save new task´s id
prwait  sys       wait              wait for child task
        cmp.l     d0,d1             right one?
        bne.s     prwait            wait some more if not
        ...                         continue code for old task
        ...
        sys       term
newtsk  sys       exec,name,args    new task probably does exec
        bra       excerr            branch if error in exec
        ...
        ...
```

In this example, the *wait* system call at *prwait* makes the old task wait for the new one (it´s child) to finish before continuing. Note that the *wait* system call returns the terminated task´s id in the D0 register.

4400 FILE HANDLING

This topic describes the manipulation of files, console, directories, printers, and other devices on the 4400.

General File Definitions

Before delving into the actual manipulation of files on the 4400, we need to define and describe some of their characteristics.

Device Independent I/O

Under the 4400 operating system, anything outside the program's memory, which the program can write to or read from, is treated the same way. A file on disk, a terminal, a pipe, and a printer spooler are treated the same way. This concept, termed *device independent I/O* means you can develop a program that sends its output to a terminal, and that same program, without change, will also be able to output to a disk file, printer spooler, pipe, or any other device on the system. This feature lends a great amount of versatility to the system and makes program development and updating much smoother.

This device independence is made possible by device driver routines — the system routines that take care of the specifics of the device for which they are written, creating a standard interface to the device. There is a routine to open the device and one to close it. These permit the system to do anything necessary to prepare the device for reading and writing or to finalize anything necessary when all I/O is complete. The two most important device driver routines are the *read* and *write* routines, which permit the caller to read or write data from the device.

File Descriptors

A *file descriptor* informs the system which file to operate on. (We use the term *file*, but because of device independence, the file descriptor can refer to a disk file, terminal, pipe, or any other device). The file descriptor is a four-byte numeric representation of a specific file or device. This number is assigned to the file by the system when that file is opened or created. The operating system then keeps track of the file descriptors and the files to which they are assigned. In this way, the user supplys a number instead of an entire file name each time the file is to be referenced.

For example, the *read* system call requires a file descriptor value in the D0 register before making the call. In general use, we would have saved the file descriptor number of the file we wish to read when it was opened. Now, to do the read, we need only load the D0 register with that number.

File descriptor numbers begin with 0 and extend up to the maximum possible number of open files on the system per task. This maximum will vary depending on the system configuration, but generally will be around 20-30.

Standard Input and Output

When the shell begins execution of a task, it automatically assigns input and output files to that task. Generally the input file is the user's keyboard, and the output file is the user's display. In fact, when a task begins execution, it can always count on three input/output files being already opened, assigned a file descriptor, and ready for reading or writing: *standard input, standard output,* and *standard error output.* Standard input is an open file ready for reading and is always assigned a file descriptor of 0. Generally the standard input file is the 4400 keyboard. Standard output is an open file ready for writing to and is always assigned a file descriptor of 1. Generally the standard output file is the 4400 display. Standard error output is an open file ready for writing to and is always assigned a file descriptor of 2. This output file is reserved for reporting error messages. Standard error output is initially the 4400 display.

Because these standard input and output files are already opened and assigned a file descriptor, the user program does not have to perform any *open* or *create* calls in order to perform I/O activities on them. As soon as a task begins running, it can perform a read with a file descriptor of 0 (standard input) or write with a file descriptor of 1 or 2 (standard output and error output).

Standard input, output and error can be *redirected* without any change to the program. In other words, a program which outputs some message to the user's terminal can also output the message to a disk file without any modifications. This I/O redirection is accomplished from the shell by use of the "<", ">"and "^" operators (redirected input, output and error, respectively). If the shell desires, it can provide a standard input, output or error file to the program which is different from the user's terminal. The user program need not be concerned with what the standard input, output or error is pointing to. Because of device independence and the fact that the program knows that the file or device (whatever it may be) has been previously opened, the program simply performs the I/O and doesn't care where it's going.

Opening, Closing, and Creating Files

Before a file or device can be read from or written to, it must be opened. When a program has completed all its input and output to a file, it should generally close that file. A user program may also need the ability to create new files on the system. This addresses those operations in detail.

The *open* System Call

The format of an *open* system call is:

```
        sys open,fname,mode
```

The *fname* is a pointer to a zero-terminated string containing the name of the file to be opened. The *mode* is a number (0, 1, or 2) which sets the read/write mode. If 0, the file is opened for reading only. If 1, the file is opened for writing only. If 2, the file is opened for both reading and writing.

On return from the open call, register D0 contains the 4-byte file descriptor number assigned to that file. All future references to the file is made via this file descriptor.

An error is returned from this call if the file to be opened does not exist, if the task opening the file does not have proper permissions, if too many files are already opened, or if the directory path leading to the file cannot be searched.

The *close* System Call

When a task terminates, the operating system automatically closes any files that remain open. It is wise, however, to manually close files within a program whenever possible. There are two reasons for doing so. First, since each task has a finite number of files which may be open at one time, closing a file frees up a slot in which another file may be opened. Second, in case of a system crash, you are better off having closed any files which no longer require I/O. The *close* system call is performed by loading register D0 with the file descriptor of the file you wish to close, then performing a *sys close*.

The *create* System Call

The *create* system call is used to create disk files. Other system calls are used to create directories, pipes, devices, etc. The format of create is:

```
        sys create,fname,perm
```

Once again, *fname* is a pointer to a zero-terminated string containing the name of the file to create. The file is created in the default directory unless a directory is explicitly specified in the file name. The *perm* is a value which permits the user to set the desired permissions on the new file. (Refer to Section 4, *System Calls* for details of setting these permissions.)

Note that if the file already exists in the specified directory, it is truncated to zero length (all existing data deleted). In addition, the original permissions is retained regardless of the *perm* value supplied to the create call. In other words if the file *fname* already exists, the *perm* parameter on the create call is ignored.

If the file does not exist, permission setting is subject to any default permission settings the file owner has previously specified. The *perm* parameter in the *create* call allows you to deny permissions which the default permissions grant, but does not let you grant permissions that the default permissions deny. You can think of this as a logical AND of the *perm* parameter and the default permission byte.

Every task has associated with it a default permissions byte. If that task attempts to create any new tasks, the new tasks are created with at least those default permissions. As we saw above, additional permissions may be denied by the *perm* value specified to a *create* call. Additionally, the new task is started with the same default permission byte (for creating more tasks) as it's parent. In normal use, a user may set the default permissions in his copy of the shell upon first logging on. If the default permissions are not changed by the user or any task he runs, any files the user creates will have those default permissions. (Note that the user can change default permissions with the *dperm* command and for a task to change its own default permissions with the *defacc* system call.

Reading and Writing

Perhaps the most heavily used system calls are *read* and *write*. It is by these functions that a program communicates with the user, disk files, printers, other tasks, and anything else in the outside world. Reading and writing permits great versatility in how files are accessed. For example, with a disk file, the user can begin at any particular point in the file (right down to a specific character) and read or write as many characters as desired from that point. This makes both sequential and random access of the files quite simple.

The *read* and *write* system calls assume a *file position pointer* has already been set. This is a pointer which the system maintains to show the current position for reading and writing in a file. The discussion on *seeking,* later in this section, shows how it can be set. The only parameters required, then, are the file descriptor to specify which file, the count of characters to be read or written, and a memory buffer address to read into or write from.

The *read* System Call

To execute a *read* call, you must first load register D0 with the file descriptor number. Then you make the *read* call with the following syntax:

```
        sys read,buffer,count
```

The *buffer* parameter is an address in the user program's memory. It specifies where the data read from the file should be placed in memory. The *count* is the maximum number of characters the programmer wants the system to read. We say maximum because, depending on the situation, the system may not actually read as many characters as requested. Upon return from the read system call, register D0 contains the number of bytes that was actually read.

When dealing with a regular disk file, the system will always read *count* bytes if possible. There are only two reasons that the system would read less than that number from a regular disk file: a physical I/O error occurs, or the specified count forces the system to attempt to read past the end of the file. For example, if a file has only 120 characters and a *read* call is issued with a *count* parameter of 256, the read takes place and return with no error, but shows that only 120 characters were actually read. After this call the file position pointer is left pointing at the end of the file. Any subsequent read call returns with no error, but with the number of bytes read equal to zero. This is in fact how a user program should detect an *end of file* condition: a return from a read system call with no error but with the actual number of characters read being zero.

Reading and writing to the console display and keyboard is handled with the same system calls as when reading and writing disk files. There is a difference in the result of a read call, however, in that if the file being read is the console, only one line is returned at most. By one line we mean all the characters typed since the last carriage return, terminated by a carriage return. Thus, even though we execute a call with a desired *count* of 1024 characters to be read, if the user at the console types the letters *halt* followed by a carriage return, the read call would return with an actual-bytes-read count of only five. If the user has not typed anything when the call is issued, the calling program must wait until something is typed.

As with regular disk files, it is possible to detect an *end of file* condition from a keyboard by performing a *read* and receiving no error and no characters. An *end of file* condition from a keyboard is produced by typing a Control-D. Note that the Control-D character itself is not actually passed on to the operating system, only the *end of file* condition.

As an example of the use of the read call, let's examine a section of code that attempts to read 1024 bytes of data, placing them in a buffer called *buffer*. We assume the file has already been opened for reading and the file descriptor is stored at *fdsave*.

```
...
...
move.l   fdsave,d0           get file descriptor
sys      read,buffer,1024    read 1024 bytes into buffer
bes.l    rderr               branch if error
tst.l    d0                  end-of-file-condition?
beq.l    endof               special handling if so
add.l    #buffer,d0          point to end of data
move.l   d0,bufend           save buffer end pointer
...
...
buffer   ds.b                1024
...
...
```

Upon return from the *read* system call, we first check for a returned error status. If an error occurred, we assume the program handles it properly at *rderr*. If no error, we check for an *end of file* condition. Recall that an *end of file* condition is recognized by a program as zero characters read when there was no error. If we are at the end of the file, the program jumps to *endof*, where we again assume that such a condition is properly handled. If we did not receive an error and were not at the end of the file, our program calculates a pointer to one past the last byte read into the buffer and stores that pointer at *bufend*. Normally this pointer should be *buffer+1024*, but if the read call returned less than 1024 bytes it would be lower.

The *write* System Call

The *write* function is executed by first loading register D0 with the file descriptor number and then issuing the *write* call:

```
sys write,buffer,count
```

The *buffer* parameter is the address of the location in the user program's memory where the program writes the data. The *count* is the number of characters to be written to the file. Upon return from the *write* system call, the D0 register contains the actual byte count written (if there is no error). It is not necessary to compare this value to the requested count to be written because if there was no error, you can be sure the entire write function took place properly.

Let's look at a complete program to send the message *Hello there!* to the standard output file. If there is an error in writing to that file, we will then send the message *Error writing standard output.* to the standard error output file. (Recall that the standard output is assigned file descriptor number 1 and standard error output is assigned file descriptor number 2.)

```
        lib     sysdef          include system definitions
        text

* start of main program

sayhi   move.l                  #1,d0write to standard. output
        sys                     write,hello,hlngsend message
        bec.s                   doneexit if no error
        move.l                  d0,-(a7)else, save error number
        move.l                  #2,d0write to std. error output
        sys                     write,erm,elngsend error message
        move.l                  (a7)+,d0restore error number
        bra.s                   done2
done    move.l                  #0,d0
done2   sys                     termterminate program

* strings

hello   fcc                     'Hello there!',$d,0
hlngequ                         *-hellocompute length of string
erm fcc                         'Error writing standard output.',$d,0
elngequ                         *-ermcompute length of string

        end                     sayhigive starting address
```

There is no *open* system call because we know that the standard output and standard error output files are already opened and ready for writing when the program begins execution. Note the convenient method of providing the count of characters to be written. Also note that we did not look for an error after the system call to write to the standard error output. We really have no good recourse if an error does occur while reporting an error, so we simply terminate.

Efficiency in Reading and Writing

There are several things a system programmer can do to achieve efficient reading and writing of files on the 4400. The first and most obvious of these is to read or write as much of a disk file as possible with a single call. There is much less system overhead in executing one call to read 4096 characters than in executing 32 calls to read 128 characters each. The most efficient reads and writes are those made in multiples of 512 bytes. This is, of course, due to the fact that the 4400 disk block size is 512 bytes. Due to the way memory mapping works, additional efficiency can be gained by placing all read and write buffers on 512 byte address boundaries in memory.

By all means do not perform single character I/O with system calls for each character. If single-character I/O is required, the user program should handle the necessary buffering so that system calls are made only on a buffer full of characters.

Seeking

For each open disk file, the operating system maintains a pointer that indicates the current position for reading or writing in that file. This pointer can point to any place in the file, right down to any specific character position. The user does not have direct access to this pointer, but may use the *seek* system call to position it to any desired spot in a file. The format of the seek call is:

```
        sys seek,offset,type
```

Before making a system call to *seek*, the user must load the desired file descriptor in register D0. Seeks are done on a relative basis. That is, a seek amount is supplied to the call and the seek is to be that amount relative to some reference point. (This reference point is the *type* parameter shown above.)

There are three possible reference points: the beginning of the file, the current position in the file, and the end of the file. The *type* value should be as follows:

type	**starting position or reference point**
0	beginning of the file
1	current position in file
2	end of the file

The argument *offset* is a four-byte 2's complement offset that represents the amount of offset to be added to the reference point to find the new position in the file. A positive number indicates forward in the file; a negative number indicates backward into the file. On return from the *seek* call, the new current position is left in register *D0*. This is the current position relative to the start of the file. To find the current position in a file, you could use a system call of *sys seek,0,1*, finding the result in *D0*.

As an example, let's construct a simple random access routine. Assume we have a data file with fixed-length records of 256 characters per record. We know we will never have more than 32000 records in our file, so the record number can be represented in 16 bits. We want to write a subroutine that will read the record specified by the record number in register *A0* and leave the data at the location specified by the *A0* register. The basic procedure will be to find the starting position of the desired record in the file by multiplying the record number by the record size of 256. Then we seek to that position and read 256 bytes. Our routine looks like this:

```
        ...
        ...
getrec  move.l  a0,iread+2   save address for read
        ext.l   d0           make record number long
        lsl.l   #8,d0        record*256 is offset
* seek to record
        move.l  d0,Iseek+2   set seek address parameter
        move.l  fd,d0        assume file descriptor at fd
        sys     ind,Iseek    indirect call to seek
        bes.l   skerr        branch if error
```

```
* file pointer positioned, now read record
        move.l  fd,d0         get file descriptor
        sys     ind,iread     indirect call to read
        bes.l   rderr         branch if error
        rts                   all finished

        ...
        ...
Iseek   dc.w    seek          seek function
        code
        dc.l    0             seek address (unknown)
        dc.l    0             type 0: position from begin
iread   dc.w    read          read function code
        dc.l    0             buffer location (unknown)
        dc.l    256           character count to read
        ...
        ...
```

Notice that we used indirect calls to *seek* and *read,* because at assembly time we do not know what address we will need to seek nor where in memory to place the data we read. By using indirect calls, we can set aside an area of memory (at *Iseek* and *iread*) where these values can be stored as the program executes.

File Status Information

The *status* and *ofstat* calls are used to obtain information about each file or device. *Ofstat* is used to obtain information about a previously opened file while *status* obtains information from an unopened file. The format for *ofstat* is:

```
<file descriptor in D0>
sys ofstat,buffer
```

The user must load register D0 with the file descriptor of the previously opened file.

The format for *status* is:

```
sys status,fname,buffer
```

With *status*, the file is specified by providing the *fname* parameter, which is a pointer to a zero-terminated string containing the desired file name. In both commands the *buffer* parameter is a pointer to a buffer in memory or an area of memory into which the information about the file can be placed. This buffer must be at least 22 bytes long. When the *status* or *ofstat* call is completed, this buffer contains all the information available for the file in the format described below.

Assuming the buffer begins at some location called *buf*, the information in the buffer is:

Name	Location	Field Size	Information in Field
st_dev	buf	2	device number
st_fdn	buf+2	2	fdn number st_fil
	buf+4	1	spare (for word alignment)
st_mod	buf+5	1	file mode
st_prm	buf+6	1	permission bits
st_cnt	buf+7	1	link count
st_own	buf+8	2	file owner's user id
st_siz	buf+10	4	file size in bytes
st_mtm	buf+14	4	time of last file modification
st_spr	buf+18	4	reserved for future use

The device number is a number assigned to the device on which the file resides. The fdn number is the number of the *file descriptor node* associated with the file. The file descriptor node is a block of information about the file and where it resides on the disk. It is from the fdn that *status* and *ofstat* obtain their information.

The link count is the number of directory entries that are linked to the fdn or actual file. More information on linking can be found later in this section in the discussion titled *Directories and Linking*. The file owner's user id is a two-byte id that was assigned to the user by the system manager when the user was given a user name. The file size in bytes is the exact number of characters in the file. The time of last modification is the internal representation of the last time the file was written to.

The file mode and permission bytes each hold several bits of information. This is done by assigning single bits within the file mode to particular file types and within the permission byte to the various possible permission types. The state of the particular bit (0 or 1) indicates which type of file mode or whether permission is given or denied. The File Mode byte is shown in Figure 2-1.

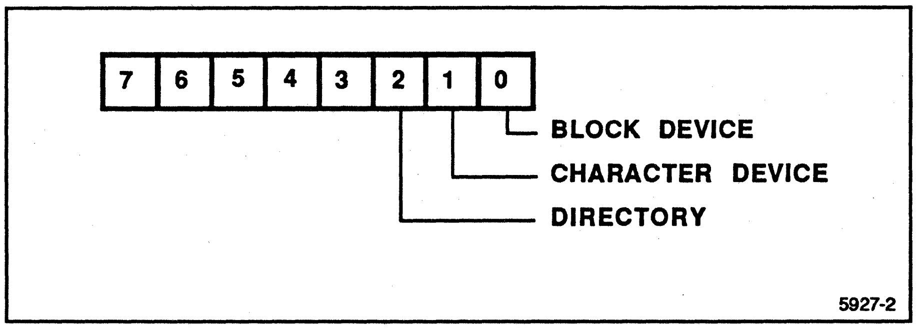

Figure 2-2. File Mode (st_mod).

Only one bit should be set at a time and it indicates the file type. A block device is a device such as a disk drive which handles data in 512 byte blocks. A character device is one such as the communications device (/dev/comm) that handles data single character at a time.

The permissions byte shows what permissions are granted or denied for the file. The Permissions byte is shown in figure 2-2.

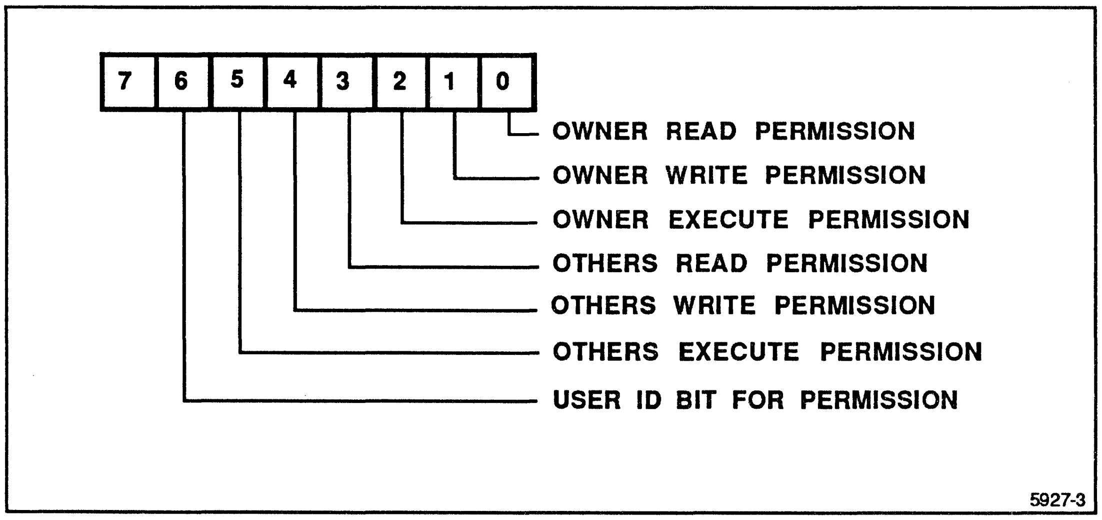

Figure 2-3. Permissions (st_prm).

In this byte, any or all of the permission bits may be set at one time. If a bit is set, that type of permission is granted. If cleared, permission is denied.

The *user id* permission bit requires further clarification. If this bit is set, it gives the user of a file the same permissions as the owner while that file is executing. As an example of the usefulness of this feature, consider a user, *joe*, who has a database program which manipulates a large data file. Now *joe* does not want anybody on the system to be able to directly read or write his data file, so he denies read and write permissions on that file to others. (Of course, he grants read and write permissions for himself.) Even though he does not want anyone to be able to read and write his data file directly, *joe* would like for other users to be able to run his database program, which manipulates the data file. All he need do is set the *user id* permission bit in his database program. With the *user id* bit set, anyone who runs the database program has the same permissions as *joe,* which allows them to manipulate the data file while running the database program. As soon as the database program is terminated, however, the other user no longer has the permissions of *joe,* the owner.

Another example of the use of the *user id* bit can be seen in the *crdir* or *create directory* command. A directory is a special type of file, and the only way to create a directory is with the *crtsd* system call. That call may only be executed by the system manager. Without the *user id* bit, the only person who could use the *crdir* command (which contains a *crtsd* system call) would be the system manager. The *crdir* program has the *user id* bit set, however, so that anyone who runs it temporarily has the same permissions as the owner. The owner of *crdir* is the system manager; thus any user can create a directory.

DIRECTORIES AND LINKING

A directory entry is nothing more than the name of a file and a single pointer to the file descriptor node (fdn) for the file. This fdn is a small unit on the disk; it contains various information about a particular file. There is one and only one fdn on a disk for each file which resides on the same disk. It is possible, however, to have more than one directory entry point to the same fdn. Two different users could have an entry in their own directory which pointed to the same fdn and therefore the same file. This feature is called a *link* and you can see it is possible to have many *links* to the same file.

A long directory listing (dir +l) shows the number of directory entries which point to or are linked to each file. This is always *1* or greater; if it ever goes to zero no one is linked to the file and it will be deleted. In fact when you *remove* a file, the command merely removes that name from the directory. This decrements the link count in the associated fdn. If that count is still non-zero, someone else is linked to the file and it is not deleted from the disk. If the count does go to zero, no one else is linked to the file and it is deleted.

An example of linking can be seen in every directory on a 4400 disk. Recall that there are two entries, . and .., in each directory. (They don't appear in a *dir* listing unless you use the *+a* option.) The . entry represents the directory in which that entry is found; .. represents the parent directory of the directory in which it is found. Thus typing . as a directory name is equivalent to typing the entire path name for the current directory. Typing .. is equivalent to typing the path name for the parent directory of the current directory. These directory entries are not separate files, but are links to the current directory file and the parent of the current directory. That is why you see a link count of more than one for every directory on the system.

The *link* and *unlink* system calls allow the programmer to link to files and unlink from files, respectively. The *link* function is quite straightforward: one specifies a pointer to the name of the file to be linked to, and a pointer to the new name that will be put into the directory. The *unlink* call is equally straightforward: the programmer simply provides a pointer to the filename or directory entry to be unlinked. This *unlink* call is the method of deleting files, the *remove* command calls on the *unlink* function to perform the file deletion. Note that a file is not deleted by an *unlink* call unless the call removes the last link to the file.

If a file is open when an *unlink* call is made, the link is removed, but the file will not be deleted or closed by the operation. The user can still read or write to the file as long as it is left open. The 4400 operating system waits until the file is actually closed and then checks the link count to see if it should be deleted from the disk. This creates interesting possibilities for a program. A file can be opened and then immediately unlinked. As long as the program leaves that file open, it can read from it or write to it. When the program is finished with the file, it has only to close it. If no one else is linked to the file, it is immediately deleted.

OTHER SYSTEM FUNCTIONS

This discussion describes several features and functions available to the system programmer that are somewhat specialized. Specific calling formats and parameters will not always be given; for this refer to Section 4, *System Calls.*

The Memory Management Functions

Earlier, we learned that when a task is started, it is allocated text, data, and stack memory according to the program size. The system automatically increases the stack size if necessary. With the *break* and *stack* system calls, it is possible for a running task to change the amount of memory allocated to it´s data or stack spaces. It is also possible to relinquish allocated memory back to the system, that is to deallocate data or stack memory. The *memman* system call controls the activity in a region of memory, and the *phys* system call permits access to certain system resources.

The *break* Function

The means of performing this dynamic memory or stack allocation and deallocation are the *break* and *stack* commands. An address is supplied to *break* and the system attempts to allocate memory to be sure there is RAM up through the specified address. Memory is allocated in page sized sections, so depending on the address specified there may be some memory beyond the address. If an address is specified which falls below the amount of program memory already allocated, that memory is relinquished or returned back to the system.

New memory pages are not necessarily allocated to a task when the size of it´s address space is increased using the *break* command. Instead, new memory pages are allocated only when the program actually reads or writes to a location within such a page. Thus it is possible for a task to manage its memory by using the *break* to set the address space to the maximum size, then "touching" those pages which it actually use.

The *memman* Function

The *memman* function allows individual pages of memory to be managed. These regions may be disabled or enabled for writing, and locked or unlocked. All resources associated with a page can be released, causing the system to effectively forget information stored in the page.

The *phys* Function

The *phys* system call makes shared system resources addressable within the address space of the task. The most commonly accessed resources is the display bit map. Writing to this resource results in visible changes upon the console display.

The *ttyset* and *ttyget* Functions

The 4400's *ttyset* and *ttyget* functions provide a way to alter and examine several configuration parameters of devices. The exact nature of these parameters differ for the various devices. For example, the console device includes parameters such as the line-cancel character, the backspace character, mapping of upper to lower case, tab expansion, etc. For any device the parameters are represented in six bytes of data. These six bytes can be read with the *ttyget* system call to examine the current configurations, or can be set with the *ttyset* system call to alter the current configuration. A six-byte buffer must be established in memory to hold the desired configurations for *ttyset* or to receive the current configuration information for *ttyget.* The file */lib/systty* contains definitions for the structures and constants.

Console Device parameters

If we assume that a six-byte buffer called *ttbuf,* the data has this format:

Name	Location	Contents
tt_flg	ttbuf	Flag byte
tt_dly	ttbuf+1	(reserved)
tt_cnc	ttbuf+2	Line cancel character (default is Ctrl-U)
tt_bks	ttbuf+3	Backspace character (default is Ctrl-H)
tt_spd	ttbuf+4	Terminal speed
tt_spr	ttbuf+5	Stop output byte

The eight bits of the Flag byte represent eight different modes of operation for the console. When set, they imply that the indicated mode is in operation. The format of the Flag byte is shown in figure 2-3.

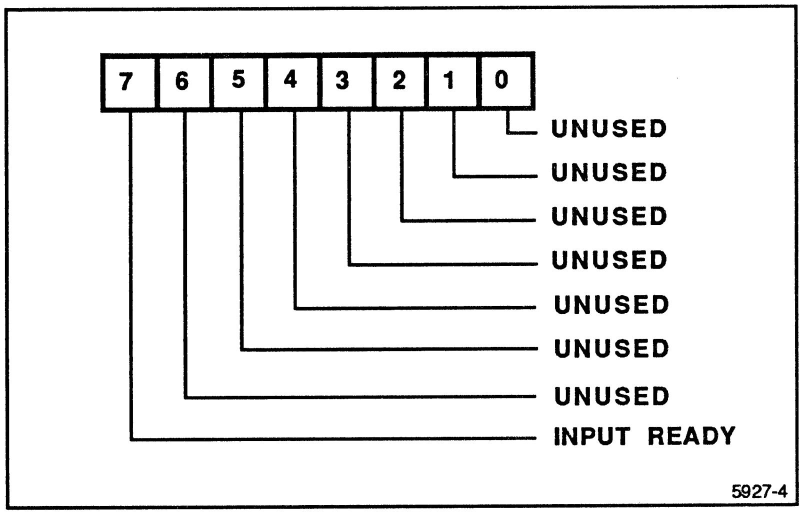

Figure 2-4. Flag Byte (tt_flg).

When set, *Any Character Restarts Output* bit instructs the console driver to restart the output if it has been stopped by either an escape or XOFF.

The Terminal Speed byte presently implements only one bit. It is the high order bit (bit 7) and, if set, indicates that the terminal has input characters waiting for the program. This bit is meaningful only when read, i.e. the input-ready condition cannot be set via this bit and *ttyset*. The format of the Terminal Speed byte is shown in figure 2-4.

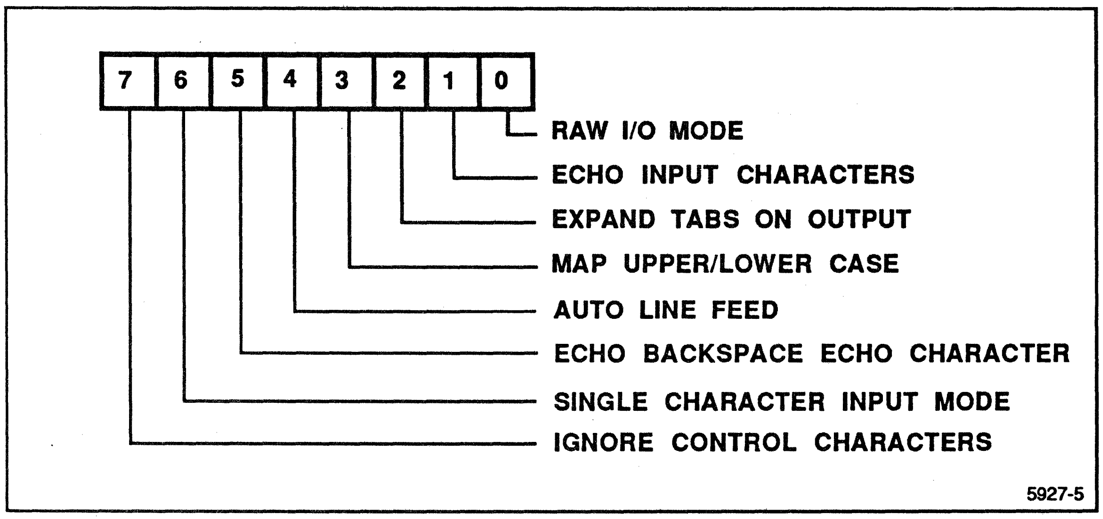

Figure 2-5. Terminal Speed Byte (tt_spd).

Under normal input operations, the *Input Ready* bit is not set until an entire line has been input and terminated by a carriage return. There are special input modes which can be established, however, where the *Input Ready* bit will be set as soon as a single character is input. These are the *raw I/O mode* and the *single character input mode*, described later in this section.

The Stop Output byte contains bits which control the stopping and starting of output to the console. There are two methods by which a user can stop and start output to the console: the escape key and XON/XOFF processing. The escape key method permits a user to type an escape character (hex 1B) to stop output. A subsequent escape character restarts the output. The XON/XOFF method permits a user to type an XOFF character (hex 13) to stop output and a subsequent XON character (hex 11) to restart it. The escape and XON/XOFF mechanisms can be independently enabled or disabled by setting or clearing the proper bits in the *tt_spr* byte. The format of the Stop Output byte is shown in figure 2-5.

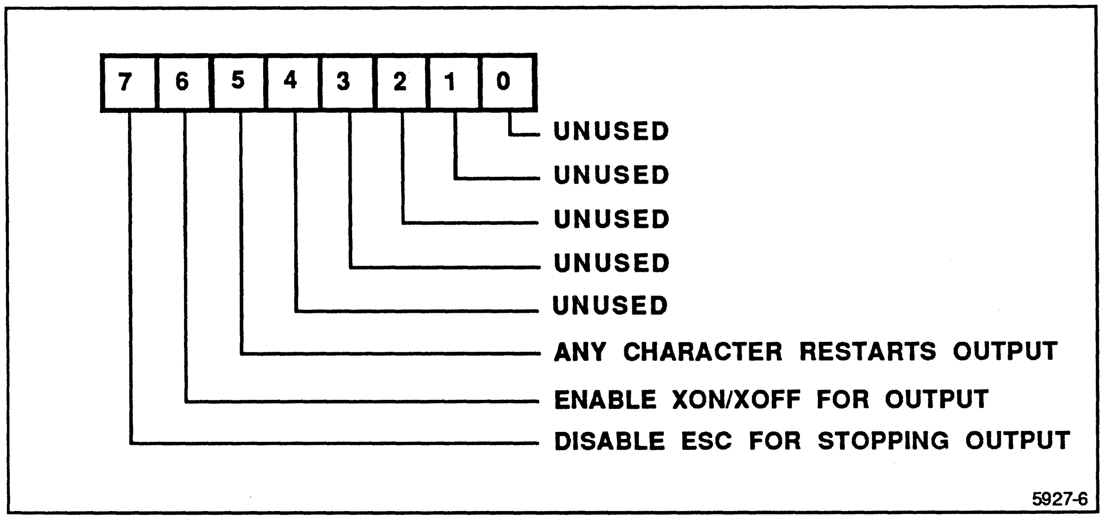
7
6
5
4
3
2
1
0
UNUSED
UNUSED
UNUSED
UNUSED
UNUSED
ANY CHARACTER RESTARTS OUTPUT
ENABLE XON/XOFF FOR OUTPUT
DISABLE ESC FOR STOPPING OUTPUT
5927-6

Figure 2-6. Stop Output Byte (tt_spr).

The following paragraphs describe each of these modes.

Raw I/O Mode

In *raw mode*, the console driver effectively does no special processing of the input or output characters. Each and every character typed on the console is directly input, including backspace characters, line cancel characters, tab characters, Ctrl-C characters, and so on. Similarly, every character output to the console is output directly: no tab expansion is performed, no line feeds are appended to carriage returns, etc. In addition, the parity bit is not stripped on either input or output.

In *raw mode*, the executing program has complete control of every character input or output and the program must perform any special processing itself. Under raw mode a *read* system call will not have to wait for an entire line to be input before it can read characters. If there is a single character available, the *read* call returns with just that character. It is still possible for a single *read* call to read more than one character, but only if the characters have already been typed into the input buffer before the call is made. This mode is off by default.

Echo Input Characters

If this mode is enabled, each character typed on the keyboard is echoed to the display console. An example of this mode occurs when a user logs in and is asked for his password. The login program writes the *Password:* message and then turns the *echo input characters* bit off while the password is entered. In that way the password is not echoed to the screen. This mode is on by default.

Expand Tabs on Output

If the terminal does not have hardware tab expansion, this bit can be set to allow the terminal driver software to automatically expand tabs on output. Tab stops are assumed to be at 8 column intervals. In other words, if this bit is on, then each time a horizontal tab character ($09) is output, the system spaces over to the next column which is a multiple of 8 (unless it is already at such a column). This mode is off by default.

Auto Line Feed

When this mode is on, the console driver will automatically output a line feed ($0A) after each carriage return is output. This mode is on by default.

Single Character Input Mode

Single Character Input Mode allows a program to input one character at a time without having to wait for a carriage return. When not in the single character input mode, a call to read a single character would have to wait until an entire line terminated by a carriage return had been typed before it would have access to a single character within the line. If single character input mode is on, the program can read a character as soon as it has been typed. Note that it is still possible to read multiple characters while in the single character input mode, if they are available. While in the single character input mode, the parity bit is stripped off of input characters, but only Ctrl-C, Ctrl-D, and Ctrl-\ are treated as special characters. In other words, tabs, backspaces, and line cancels are ignored and should be processed by the user's program if desired. This mode is off by default.

Ignore Control Characters

When this mode is on, the system ignores all control characters except for the following:

- Carriage Return
- Horizontal Tab
- Ctrl-C
- Ctrl-D
- Ctrl-\
- Backspace Character
  (if defined to be a control character)
- Line Cancel Character
  (if defined to be a control character)

Those control characters that are ignored will still be echoed if the echo input characters mode is also on. This mode is off by default.

Communications Device Parameters

The communication device parameters use the *ttyset* and *ttyget* system calls to communicate option settings to the communications port device driver. The format of the 6-byte buffer used with these calls is defined differently than for standard tty devices. The file */lib/syscomm* contains definitions for the structures and constants.

If you call a six-byte buffer *cbuf*, then the following data is in this format:

Name	Location	Contents
c_com	cbuf	Command field
c_value	cbuf+1	Additional values
c_parity	cbuf+2	Parity selection
c_flag	cbuf+3	Flow control
c_ospeed	cbuf+4	Output baud rate
c_ispeed	cbuf+5	Input baud rate

The *c_com* field is used to request various commands to be executed by the device driver during *ttyset* and *ttyget* calls. Valid values for this field are defined as:

Command	Value	Description
RESET_COMM	1	Reset the communications port
SETUP_COMM	2	Set parity, flags and baud rates
EXCL_COMM	3	Do not accept open request until closed or reset
BREAK_COMM	4	Send break signal for c_value tenths of a second
NOBLOCK_COMM	5	Read calls do not block
BLOCK_COMM	6	Read calls do block (default)
DTRLOW_COMM	7	Set DTR signal low
DTRHIGH_COMM	8	Set DTR signal high (default)
RTSLOW_COMM	9	Set RTS signal low
RTSHIGH_COMM	10	Set RTS signal high (default)

The RESET_COMM command resets the */dev/comm* device to its default conditions.

The SETUP_COMM command causes:

- Parity type and number of stop bits to be set according to the value in the *c_parity* byte
- Flagging control to be set according to the *c_flag* byte
- Baud rate to be set according to the *c_ospeed* and *c_ispeed* bytes

The EXCL_COMM command prohibits another process from opening this device until it is closed or reset.

The BREAK_COMM command sends a break signal whose length in microseconds is the value in the *c_value* byte.

The NOBLOCK_COMM command lets a read call return from the */dev/comm* device when there is no data. The read calls will not block and a zero count is returned if no bytes are available.

The BLOCK_COMM command is the default state and reads will block if no data is available. If data is available, the data is read into the caller's buffer (up to the requested number of bytes) and the number of bytes read is returned.

The DTRLOW_COMM and DTRHIGH_COMM commands set the *Data Transmit Ready* signal low or high, respectively.

The RTSLOW_COMM and RTSHIGH_COMM commands set the *Request To Send* signal low or high, respectively.

The *c_parity* byte selects both the parity and the number of stop bits. Valid values for this field are defined as:

Condition	Value	Description
LOW_PARITY	0	Parity bit always 0
HIGH_PARITY	1	Parity bit always 1
EVEN_PARITY	2	even parity
ODD_PARITY	3	odd parity
NO_PARITY	4	no parity (default)
TWO_STOP_BITS	0x80	if most-significant-bit set, then two stop bits, else one stop bit

The *c_flag* byte selects the type of flow control to be used through the communications device. The valid values are:

Condition	Value	Description
NO_FLAG	0	No control flagging
INPUT_FLAG	1	Send ^S/^Q for input control
OUTPUT_FLAG	2	Accept ^S/^Q for output control
TANDEM_FLAG	3	Use both input and output ^S/^Q (default)
DTR_FLAG	4	Use DTR/CTS for flow control

By default, read calls will block if no input is available. If any data is available, it is read into the caller's buffer (up to the requested number of bytes) and the number of bytes read is returned. If NOBLOCK_COMM is requested, then read calls do not block and a zero count is returned if no bytes are available.

The following constants are used in the *c_ospeed* and *c_ispeed* fields to indicate the transmit and receive baud rates:

Constant	Value
EXTERNAL	0
C50	1
C75	2
C110	3
C134	4
C150	5
C300	6
C600	7
C1200	8
C1800	9
C2400	10
C4800	11
C9600	12
C19200	13
C38400	14

Pseudo Device Parameters

A pseudo terminal (pty) is a pair of character devices, a master and a slave device, which provide an interface identical to that described in */lib/systty* and */lib/include/sys/sgtty.h.* While other devices have a hardware device of some sort behind them, the slave device has instead, another process passing data through the master half of the pseudo terminal. Anything written on the master device is given to the slave device as input and anything written on the slave device is presented as input to the master device.

The system supports 32 pseudo terminal pairs, named /dev/pty00 through /dev/pty31. The system call *create_pty* returns two file descriptors; the master pty in register A0, the slave pty in register D0. Both a slave and master device are opened by a *create-pty* call. The slave device can be closed and reopened again by name, provided the corresponding master device is still open.

Pseudo terminals can use normal tty calls, ttyset and ttyget. Also, pseudo-terminal information can be set/returned from the master side of a pty with the *control_pty* system call. The file */lib/syspty* contains definitions for the structures and constants.

The system call *control_pty* is used to control the behavior of a pseudo-terminal channel. The command structure is:

```
<master device file descriptor in D0>
sys control_pty,function,cval
<state in D0>
```

The functions are:

Function	Equ	Description
PTY_INQUIRY	0	Return the state of the channel
PTY_SET_MODE	1	Change the control mode of the channel
PTY_FLUSH_READ	3	Clears data queue on output of master
PTY_FLUSH_WRITE	4	Clears data queue on output of slave
PTY_STOP_OUTPUT	5	Prevents slave from writing to master
PTY_START_OUTPUT	6	Allows slave to write to master

All of the functions return the state of the channel in register D0 as described by the function *PTY_INQUIRY*. The function *PTY_INQUIRY* is used to return the state of the channel. For this function, *cval* is ignored. The value returned is a combination of bits which describe the state of the channel. The bits are:

Mode	Bit	Description
PTY_PACKET_MODE	0	Reads status on master
PTY_REMOTE_MODE	1	No edit of data to slave
PTY_READ_WAIT	2	Block on non-satisfied reads
PTY_WRITE_WAIT	3	Don't block on writes
PTY_HANDSHAKE_MODE	4	Remote writes not satisfied until consumed
PTY_SLAVE_HOLD	7	Prohibit slave from writing data
PTY_EOF	8	No more slave connections
PTY_OUTPUT_QUEUED	9	Slave has some output queued
PTY_INPUT_QUEUED	10	Slave has some input queued

These bits are defined as:

PTY_PACKET_MODE	When packet mode is selected on the master side of a pseudo terminal, reads on the master side return two bytes of status in addition to any data written by the slave. If any slave data is available, the status bytes are zero. If no data is present, the status bytes are the same as those returned by *PTY_INQUIRY*. Set to 1 if selected.
PTY_REMOTE_MODE	If this bit is set, data written by the master and sent to the slave side, will be flow controlled with no editing. Set to 1 if selected.

PTY_READ_WAIT	If this bit is set, a read on the master side is blocked until slave data is available. If this bit is clear, read requests on the master pseudo terminal return regardless of whether data is available. Set to 1 if selected.
PTY_WRITE_WAIT	If this bit is set, the master pseudo terminal does not hang on a write request if the output buffer is full. Set to 1 if selected.
PTY_HANDSHAKE_MODE	If this bit is set, a write on the master pseudo terminal is not complete until the slave has consumed the data. Set to 1 if selected.
PTY_SLAVE_HOLD	If this bit is set, the slave pseudo terminal is prohibited from writing any more data to the channel. Set to 1 if slave pseudo terminal's I/O is stopped.
PTY_EOF	Set to 1 if all slave ptys associated with this pty channel are closed.
PTY_OUTPUT_QUEUED	The slave pseudo terminal has written data to the channel which has not yet been consumed by the master. Set to 1 if output queued on the slave device.
PTY_INPUT_QUEUED	The master pseudo terminal has written data to the slave side which has not yet been consumed by the slave pseudo terminal. Set to 1 if input queued on the slave device.

The default mode of a pseudo device channel when created by a create_pty call is all modes set to zero.

The function *PTY_SET_MODE* is used to change the control mode for the pseudo-terminal channel. The value *cval* contains the new mode and should be some combination of the bits described in the previous section. The new control mode is exactly what is in *cval* so to perform an incremental change, the current value must be obtained using *PTY_INQUIRY*.

The function *PTY_FLUSH_READ* causes any data written by the master side to the slave input queue to be purged.

The function *PTY_FLUSH_WRITE* causes any data written by the slave side that has not yet been consumed by the master side to be purged.

The function *PTY_STOP_OUTPUT* prevents the slave side from writing any more data to the master side. This condition is reflected in the status bit *PTY_SLAVE_HOLD*.

The function *PTY_START_OUTPUT* allows the slave side to continue writing data to the master side.

Pipes

A *pipe* is a mechanism that permits a task to communicate with a child task.

A *pipe* allows communication in one direction only; it allows one task to send information to another, but not to receive. If a pair of tasks need two-way communication, two pipes must be established; one to send from the first task to the second and one to send from the second task to the first. Once the *pipe* is established, the first task sends information to the second by using the *write* system call, just as it would in writing to any other device. The second task receives information from the first by using the *read* system call. The file descriptor numbers for these write and read operations are provided by the system when the pipe is created.

The *pipe* mechanism works sort of like a holding tank with a valve on the input and output lines. If the tank is not full, the writing task can pump data into it even though the reading task has the output valve closed (is not actively reading). Likewise, if the tank is not empty, the reading task can drain information out of it even though the writing task has the input valve closed (is not currently writing). If the tank is full, the writing task is forced to wait until the reading task has emptied it before being permitted to pump in more data. If the tank is empty, the reading task must wait until the writing task has pumped in some data. This *holding tank* is a 4K disk buffer. There is a buffer for each pipe, but none show up in any directory. These pipe buffers are placed on the disk unit which has been configured as the pipe device.

The following section of code establishes a *pipe* between a task (A) and its child task (B). First, Task A calls *crpipe* to create the *pipe*. Next, we immediately fork to create Task B, and then set up the file descriptors so that we will be writing from task A to task B. The code looks something like this:

```
        ...
        ...
        sys      crpipe         create pipe system call
        bes.l    piperr          branch if error
        move.l   d0,rdfd        save read file descriptor
        move.l   a0,wrtfd       save write file descriptor
        sys      fork           fork to spawn task B
        bra.s    child          new task B here
        bes.l    frkerr         task A checks for error
        move.l   d0,tskBid      save task id of child
        move.l   rdfd,d0        pipe read file descriptor
        sys      close          close read (A only writes)
        move.l   wrtfd,pipefd   save pipe write file descriptor
* now Task A can write to pipe using pipefd
        ...
        ...
        sys      term           end of task A
* code for Task B
child   move.l   wrtfd,d0       pipe write file descriptor
        sys      close          close write (B only reads)
        move.l   rdfd,pipefd    save pipe read file descriptor
* now Task B can read from pipe using pipefd
        ...
        ...
```

Notice that each task closes the portion of the pipe that it cannot use. As previously stated, a pipe allows data to be transmitted in only one direction. After performing the fork, both tasks have open read and write pipe files. Now it is assumed that the writing task will eventually close the write pipe file, and the reading task will eventually close the read pipe file. However, we must be sure that the writing task closes the read file and the reading task closes the write file. In fact, these files should be closed as soon as possible, before any reads or writes to the pipe are performed.

Program Interrupts

Program interrupts provide a way to interrupt tasks under software control. One program or task can send a program interrupt to another task. This permits timing and synchronization among the tasks in the system. It also gives the programmer the ability to terminate tasks prematurely under software control.

Sending and Catching Program Interrupts

Here is an example of how a program sends an interrupt.

```
...
...
move.l   #327,d0          get task number in D0
sys      spint,SIGQUIT    send quit interrupt
bes.l    error
...
...
```

Assuming the effective user id of the task executing the above code matches that of task number 327 or that the above task is owned by the system manager, a *quit* interrupt will be sent to task 327. (We will define the quit interrupt and other interrupts in a moment.) Notice the system call used to send program interrupts is *spint*. It is also possible for a program to send an interrupt to all tasks associated with the terminal which executed the program. Consult the *spint* description in Section 4, *System Calls* for details.

The *cpint* (for *catch program interrupt*) provides a way for a task to *catch* or intercept a program interrupt when it is received. The task may then permit the interrupt to complete its default action (usually task termination), may ignore the interrupt completely, or may take some special user-defined action.

In effect, *cpint* permits the user to set up an interrupt vector address, so that if a program interrupt is received, control is vectored to that address. The programmer may place a routine at that address which handles the interrupt in some special way. Two addresses, $000000 and $000001, are special. If the address specified for the caught interrupt is $000000, the default action of the interrupt is allowed to occur, much as if the interrupt had not been caught at all. If the address specified is $000001, the interrupt is ignored, much as if the interrupt had not even been sent. Note that no code is actually placed at these addresses. The *cpint* function recognizes them as special values and performs the indicated interrupt handling without ever jumping to or using them as real addresses. Any other address supplied to *cpint* is assumed to be a valid program memory address, and control is passed to that location. There, the programmer places the desired interrupt handling routine; this routine must be exited with an RTR instruction, so that control is resumed at the same point in the program where the interrupt occurred.

Once a program interrupt has been caught and processed, the system resets itself back to the default condition, and interrupts are no longer intercepted. Therefore, to continue catching program interrupts, the programmer must issue a new *cpint* call after each interrupt is processed.

Table 4-1 shows the program interrupts that are available on the 4400.

Table 2-1
4400 PROGRAM INTERRUPTS

Name	Number	Description	Comments
SIGHUP	1	reserved	
SIGINT	2	keyboard interrupt	
SIGQUIT	3	quit interrupt	produces core dump
SIGEMT	4	EMT $AXXX emulation int.	produces core dump
SIGKILL	5	task kill interrupt	can´t be caught/ignored
SIGPIPE	6	write broken pipe int.	
SIGBUS	7	bus fault	
SIGTRACE	8	reserved	
SIGTIME	9	reserved	
SIGALRM	10	alarm interrupt	
SIGTERM	11	task termination interrupt	
SIGTRAPV	12	TRAPV instruction	produces core dump
SIGCHK	13	CHK instruction	produces core dump
SIGEMT2	14	EMT $FXXX emulation int.	produces core dump
SIGTRAP1	15	TRAP #1 instruction	produces core dump
SIGTRAP2	16	TRAP #2 instruction	produces core dump
SIGTRAP3	17	TRAP #3 instruction	produces core dump
SIGTRAP4	18	TRAP #4 instruction	produces core dump
SIGTRAP5	19	TRAP #5 instruction	produces core dump
SIGTRAP6	20	TRAP #6 instruction	produces core dump
SIGPAR	21	reserved	produces core dump
SIGILL	22	illegal instruction	produces core dump
SIGDIV	23	divide by zero	produces core dump
SIGPRIV	24	privilege violation	produces core dump
SIGADDR	25	address error	produces core dump
SIGDEAD	26	dead child task interrupt	ignored by default
SIGWRIT	27	write to read-only memory	produces core dump
SIGEXEC	28	reserved	produces core dump
SIGBND	29	segmentation violation	produces core dump
SIGUSR1	30	user-defined interrupt #1	
SIGUSR2	31	user-defined interrupt #2	
SIGUSR3	32	user-defined interrupt #3	
SIGABORT	33	Program abort	
SIGSPLR	34	Spooler interrupt	
SIGINPUT	35	Input is ready	
SIGDUMP	36	Memory dump	
SIGUNORDERED*	42	FPU branch/set on unordered	
SIGINEXACT*	43	FPU inexact result	
SIGFPDIVIDE*	44	FPU divide by zero	
SIGUNDERFLOW*	45	FPU underflow	
SIGOPERAND*	46	FPU operand error	
SIGOVERFLOW*	47	FPU overflow	
SIGSNAN*	48	FPU signaling NAN	
SIGMILLI	62	Millisecond alarm	
SIGEVT	63	Mouse/keyboard event interrupt	

* These interrupts are produced only by the MC68881 Floating Point Co-processor.

If not caught or ignored, all of these program interrupts (except SIGDEAD) by default cause termination of the task to which they are sent. As listed above, some also produce a *core dump*. A *core dump* is a disk file which contains a mirror image of the contents of memory. Each byte in the program and stack space are written to a disk file immediately after receipt of the interrupt. This file can be examined to determine the state of memory at the time the interrupt was received. This is often useful for diagnostic purposes.

Many of the interrupts are initiated by MC68010/68020 exception processing. The cause of those interrupts can be understood by studying the documentation of the MC68010/68020 microprocessor. Certain interrupts in the list are not directly initiated by the MC68010/68020 and need further definition.

2 Keyboard Interrupt: Generated by typing a Ctrl-C on the keyboard. This interrupt terminates the foreground task of the associated terminal.

3 Quit Interrupt: Generated by typing a Ctrl-Backslash on the keyboard. This interrupt is just like the Keyboard Interrupt except that it additionally produces a core dump.

4 EMT $AXXX Emulation Interrupt: Generated by the processor when an instruction with the pattern 1010 in bits 15 through 12 is encountered.

5 Task Kill Interrupt: Always kills the task to which it is sent. A task may not catch or ignore this interrupt.

6 Write Broken Pipe Interrupt: Generated when a pipe between two tasks is broken. This occurs when the reader is closed and the writer attempts further writing.

10 Alarm Interrupt: Generated by the *alarm* system call after the specified number of seconds. Unless caught or ignored, this interrupt terminates the task.

11 Task Termination Interrupt: This interrupt is the normal means of interrupting and terminating a task. Unlike the Task Kill Interrupt, the Task Termination Interrupt may be caught or ignored.

14 EMT $FXXX Emulation Interrupt: Generated by the MC68010 when an instruction with the pattern 1111 in bits 15 through 12 is encountered.

26 Dead Child Task Interrupt: When a task terminates, it sends an interrupt to its parent task, informing the parent that the child has terminated. This interrupt is ignored by default— it must be explicitly caught by the parent in order to function. This interrupt remains enabled after it is caught and must be explicitly disabled.

27 Write to Read-Only Memory: An attempt was made to write to a section of memory that has been reserved as Read-Only by the memory management system.

29 Segmentation Violation: An attempt was made to access memory that is outside the address space allotted to a task.

30-32 User-Defined Interrupts: These interrupts are additional interrupts that a user program or set of programs may issue and catch for whatever purpose they wish.

33 Program Abort: A signal has been received to abort the program.

35 Input Ready: This interrupt indicates that data is available to the input device.

36 Memory Dump: An attempt has been made to dump memory that is being used.

62 Millisecond Alarm: When enabled, this interrupt occurs every millisecond.

63 Mouse/Keyboard Event Interrupts: This interrupt occurs when the mouse is moved or a key depressed.

On return from a *cpint* call, register D0 contains an address. This address is the address which the system was using on receipt of program interrupts. In other words, it is the address which was provided in the previous *cpint* call. This old address can be used to tell what kind of action a program was taking on receipt of program interrupts before the current *cpint* call. For example, assume we have a program that is ignoring quit interrupts. If we now issue the instruction:

```
        sys cpint,SIGQUIT,0
```

(which says to take the default action on receipt of a quit interrupt) we would find *1* returned in the D0 register. That 1 is the address which was previously being used, and we know that an address of 1 says to ignore the interrupt.

Knowing what type of program interrupt action is currently being taken can be very useful in the case where one task starts another. If one task is ignoring some particular interrupt and that task starts some new task running, the new task should usually also ignore the interrupt. Assume Program A starts Program B by doing a *fork* and *exec*. Also assume Program B normally wishes to catch keyboard interrupts (Ctrl-Cs) and process them in a special way. Program B should be written to first check how Program A was handling keyboard interrupts. If Program A was not intercepting keyboard interrupts or was catching them, Program B may go ahead and catch them and process them as desired. If, however, Program A was ignoring keyboard interrupts, then Program B should also ignore them. The code for Program B to handle all this properly would be:

```
        ...
        ...
        sys     cpint,SIGINT,1              Start by ignoring
        cmp.l   #1,d0                       Was program A ignoring?
        beq     contin                      If so, then so should we
        sys     cpint,SIGINT,handle  If not, catch it
contin  ...
        ...
```

Note that by ignoring the keyboard interrupt while checking what Program A was doing, we avoid a potential chance for a keyboard interrupt to come through and be improperly handled.

As an example of program interrupt catching, let´s examine a portion of code that would put a program to sleep for 30 seconds. The technique is to send an alarm interrupt with the *alarm* system call, then put the task to sleep with the *stop* system call. In order to catch the *alarm* interrupt and continue properly in our program, we will use the *cpint* system call.

```
         ...
         ...
         sys      cpint,SIGALRM,wake    catch alarm & goto wake
         move.l   #30,d0                delay 30 seconds
         sys      alarm
         sys      stop                  wait for alarm interrupt
         ...                            continue with program
         ...
         ...
wake     rtr                            do nothing with interrupt
         ...
         ...
```

The *cpint* system call tells the task to catch any alarm interrupts and handle them as specified by the code at *wake*. In this example the code at *wake* does absolutely nothing but return. That is because when the alarm is received we want to simply continue execution of the program where we left off (just after the *stop* system call).

Interrupted System Calls

Most system calls cannot be interrupted by a program interrupt. That is, once a system call is executing, it will finish regardless of whether a program interrupt is pending. Once that system call is completed, the user´s program then sees any waiting program interrupt. There are a few calls, however, which may be terminated by a program interrupt. In particular, those system calls which may be interrupted are *read* and *write* (if the device being read or written is a slow device such as a terminal or printer) and the *stop* and *wait* calls. A *read* or *write* call to a fast device, such as a disk file, is never terminated by a program interrupt.

If a program interrupt does get through to one of the system calls, the following action takes place. First, the system call immediately terminates, and control passes to the program interrupt handling code if the interrupt is caught. Then, when the interrupt handling code completes, control passes to the instruction immediately following the interrupted system call **and an error status is returned.** This error status is accompanied by an *EINTR* error (number 27). In this way, the program which made the system call can detect that it was interrupted and re-issue the system call if desired.

As an example, consider a program which prompts the user for a line of data from the terminal. If a program interrupt is sent to that program while a *read* system call is getting the data from the terminal, that call may be prematurely terminated; i.e. not all the data may be returned. Once the program interrupt handling code was complete, our program would continue right after the *read* call, but would show an *EINTR* error. Our program may choose to treat the EINTR error like any other and terminate with an error message. An alternative , however, would be to recognize that it was an EINTR error and loop back in our code to re-issue the prompt and the *read* system call to input the data again.

Locking and Unlocking Records

The *lrec* and *urec* system calls provide a record locking mechanism that prevents more than one task attempting to access a file at one time. A program or task can *lock* a record of data until such time as it is ready to *unlock* or release it for others to use. While that record is locked, no other task would be able to access it.

The operating system maintains a table showing what records are locked in the system. These records may be of any length, as specified by the task which performs the lock. Note that a single task may lock only one record in a file. However, other tasks can lock other records in that same file, and a single task can lock a record in more than one file at a time.

When a task issues an *lrec* call to lock some record within a file, the system first checks the locked record table to see if the calling task already has a record locked in this file. If so, any such record is unlocked before the new record lock can be made. Next, the system checks to see if the record to be locked is available or if some other task may have previously locked some portion of it. If available for locking, the system makes an entry in the locked record table and returns to the calling task. If the desired record overlaps some portion of an already locked record, the system returns with an ELOCK error. At this point, the calling program could take some appropriate action.

There are three ways for a task to unlock a record. The first is through use of the *urec* system call, which unlocks whatever record may have been locked by the calling task for the specified file. The second is by closing a file. Upon closing, any records locked by the task that opened the file are automatically unlocked. The third is by locking another record in the same file; this will automatically unlock any record which is currently locked.

Having said this, we must back up and tell you that *locking* a record does not really prevent another task from accessing it. Any program that wishes to can still read or write the data which some other program has locked in a record. In order for locking to provide the desired results, all programs must take upon themselves the responsibility of avoiding reading or writing to a locked record. This may be accomplished by attempting to lock records before reading or writing them. If the record is available, no error is returned, and we can go ahead with the read or write. If an error is returned (ELOCK error), we know that someone else already has the record locked and we should take some other action. One possibility is to put our task to sleep for a few seconds (with the *alarm* and *stop* system calls), and then try locking the record again. Proper use of the lock and unlock calls yields the same result as if locking actually did prevent another task from reading or writing. Note that locking and unlocking is not necessary in all cases, only in those where a data file is shared and conflicts can occur.

Shared Text Programs

The 4400 operating system lets you separate an assembly language program into two sections, a *text* segment for nonchanging memory or memory which is only read, and a *data* segment for memory which can be changed by writing into it. When a task runs this program, a section of memory is assigned to each segment. If a second task runs the program at the same time, the system recognizes the fact that it already has a copy of the text segment in memory and only loads the data segment into memory for the second task. The system then maps the same memory that contains the text segment for the first task into the address space for the second task when it runs. For more details on how to produce a shared text type program, refer to Section 3, *The Assembler and Linking Loader.*

GENERAL PROGRAMMING PRACTICES

This discussion covers several general programming practices that are recommended when writing assembly language programs to run on the 4400.

Starting Locations

Assembly language programs should not have specific origin addresses. Rather, the load addresses for the text and data sections of a program (as well as the stack established by the system) should be specified at load time. These addresses can be explicitly specified to the loader, but should generally assume the default values found in the file */lib/ std_env.* This file contains the proper addresses for the hardware memory manager and is automatically read by the linking-loader.

Stack Considerations

When a program begins execution, it is assigned a portion of memory to contain the program stack. The cpu's system stack pointer (register A7) is left pointing to some location within this memory. The user's program should not write into locations in memory higher than this initial stack pointer location. The passed parameters which lie directly above the stack pointer (higher in memory) may be read, but nothing should be written above the initial stack pointer location.

Hardware Interrupts and Traps

In general, a user program need not perform any hardware interrupt or trap handling. Some traps can be handled in the same fashion as program interrupts by using the *cpint* system call.

Delays

To maintain system efficiency, a user's program should not contain delay routines which tie up the processor for long periods of time. Because of task switching, a delay loop does not provide accurate timing delays anyway. The preferred method is to use the *alarm* system call followed by a *stop* system call. The program must also then use the *cpint* system call to catch the *alarm* interrupt and continue with the desired code.

System *lib* Files Provided

Several system library files are provided for the convenience of the assembly language programmer. Located in the */lib* directory, these files contain definitions for several system related calls, tables, buffers, etc. The programmer may include these definitions in his programs by simply using the *lib* instruction in the assembler. These files include:

syscomm	Ttyget and ttyset buffer layout
sysdef	System call definitions
sysdisplay	System display and event definitions
syserrors	System error definitions
sysints	Program interrupt definitions
sysstat	Status and ofstat buffer layout
systim	Time and ttime buffer layouts

An additional file is provided for use by the linking-loader. It is called by the linking loader and should not be included in an assembler program.

std_env	Standard environment for linking-loader, linked to the file *ldr_environ.*

Generating Unique Filenames

Often, it is necessary for a program to generate a filename. A typical example is when a program wishes to create a scratch file of some sort. In a single-task environment, the program could just use some name defined at assembly time. In a multi-task environment, however, more caution is required. If the program which generates the filename is run as more than one task (background/foreground for example) there may well be conflicts since each copy of the running program would be attempting to create and manipulate the same file. The proper technique to avoid this problem is to have the program include the current task id as part of the filename. Since each executing copy of the program has a different task id, they each generate different filenames. Use the *gtid* system call to obtain the task id number, then convert it to ASCII and include it as part of the filename.

Debugging

Assembly language debugging on the 4400 is accomplished via the *debug* command. This command provides tools such as memory dumps, breakpointing, and single-stepping. Refer to Section 2, *User Commands and Utilities* in the Operators Reference Manual, for documentation on the *debug* utility.

PROGRAMMING EXAMPLE

The following sample utility demonstrates several of the calls and techniques in writing assembly language utilities on the 4400. This utility reads a file (or list of files) and strips out all control characters except for carriage returns ($0d) and horizontal tabs ($09). The syntax of the command line is as follows:

```
        strip [file] ...
```

The square brackets indicate that the file name specification is optional. If no filename is supplied, *strip* reads the standard input. The three periods (...) indicate that it is possible to supply more than one file name. In such a case, strip reads all the files in order and writes the stripped output to the standard output.

Our basic task, then, is to read either a list of files or the standard input, strip the necessary control characters, and write the result to the standard output device. In order to handle any size file(s), we shall read and write the data into a buffer. We know that for efficiency, the buffer should be an even multiple of 512 bytes, but how big a multiple? The code to implement this utility will obviously be quite small, such that the program and the buffer could easily fit in 4K of memory. Since this utility will probably not be frequently used, we decided to limit the program memory utilization to only 4K. We will make the read/write buffer as large as possible within that 4K space, while keeping it a multiple of 512 bytes.

The first step, after titling and describing the program, is to include the system definitions with the *lib* instruction on line 17. Next we actually begin the code section of our program with the *text* statement in line 23. In line 27 we load the "a6" register with a pointer to the list of filename arguments. The list is null if no filename was specified. Notice that we skip eight bytes, four containing the argument count and four containing argument 0 which is the name of the command itself.

Lines 28 through 31 check to see if a file or files were specified on the command line. If so, the argument count (what the system stack is pointing to) will be greater than 1 because argument 0 (the command name) counts as one. If the argument count is 1, no file was specified, so we must read the standard input. The file descriptor for standard input is 0, so that value is saved in *ifd* and we jump ahead to process that input. If a file was specified, we enter a loop to read through all specified files.

In line 35 we obtain the pointer to the next file in the list and store it at *opname*. If that pointer is zero (a null pointer), we have reached the end of the list, and we jump off to the exit code at *done*. If it is non-zero, it must be the address of a filename string. Lines 40 through 42 open that file for read and save the file descriptor in *ifd*. Note that the open is done via an indirect system call. This is necessary because when the program is written, we do not know what filename to specify in an open call. The pointer to the name of the file to be opened is only discovered as we run the program. When we stored the filename pointer at *opname* in line 35, we were actually storing the filename pointer in the parameter list for the upcoming indirect open system call.

In line 46 we call a subroutine named *strip* to read through the file whose descriptor is in *ifd*, strip out the control characters, and write the result to standard output. Line 47 branches back to the top of the loop to look for another possible input file.

The *strip* subroutine is where the control characters are actually stripped. In lines 67 through 69 we read *BUFSIZ* characters into memory at *buffer*. Lines 73 and 74 check for end-of-file. If we were at the end of the file, we jump to *strip9* and exit the subroutine. If not, we go on to lines 80 through 91, where the control characters are stripped from the buffer. Note that after the control characters are stripped, the resulting data is left in the same buffer. Because some characters may have been stripped out, the location of the end of the data in the buffer may be lower than before the stripping.

After the stripping, we fall into lines 96 through 101, where the stripped data is written out to standard output. Lines 96 and 97 calculate the number of characters to write. It is equal to the difference between the pointer to the end of the data in the buffer and the pointer to the beginning of the buffer. The result is stored in the parameters for an indirect write call. In line 98 we obtain the file descriptor for the standard output file. Lines 99 and 100 carry out the indirect write system call. In 101 we jump back to the beginning of the subroutine to read in another buffer of data.

Lines 113 through 134 contain the error handling code. If an error occurs, we simply write an appropriate message to the standard error output (file descriptor 2). The important thing to note about this code is that we save the error status so that it may be passed on to the *term* system call.

Lines 144 through 158 contain temporary storage and buffers. First are the parameter lists for the indirect open and write calls mentioned earlier. Line 153 reserves storage space for the current input file descriptor. Lines 155 through 158 reserve the read/write buffer. The buffer starts on a 512 byte boundary and the end of the buffer is the end of the 4K memory page. Recall that read/write efficiency is gained not only by a buffer size which is a multiple of 512 bytes, but also by beginning the buffer on a 512 byte boundary. Line 157 establishes the buffer size by calculating the difference between the end of the 4K page ($1000) and the beginning of the buffer. The *end* statement on line 161 specifies the utility starting address in its operand field.

SAMPLE *strip* UTILITY

```
1 **********************************************************************
2 *
3 * Sample "strip" Utility
4 *
5 * Copyright (c) 1984 by
6 * Technical Systems Consultants, Inc.
7 *
8 * Utility to strip all meaningless control characters from
9 * input file and write stripped version to standard output.
10 * Accepts list of input files or defaults to standard input.
11 * For the purpose of this utility, "meaningless control
12 * characters" are all characters with and ASCII value between
13 * $00 and $1F inclusive except carriage return ($0D) and
14 * horizontal tab ($09).
15 **********************************************************************
16
17          lib      sysdef       read system definitions
18
19 ********************************************
20 * start of main program
21 ********************************************
22
23          text                   begin text segment
24
25 * start by seeing if any input files were specified
26
27  start   lea      8(a7),a6     set arg ptr past count & arg0
28          cmp.1    #1,(a7)      file specified only if argcnt >1
29          bhi.s    main2        branch if filenames present
30          move.1   #0,ifd       else use standard input
31          bra.s    main4        go process std. input
32
33 * check to see if any more files specified
34
35  main2  move.1    (a6)+,opname  get next argument in list
36         beq.s     done          branch if no more args
```

```

* open specified file for read

       sys       ind,iopen    do indirect open call
       bes.s     opnerr       branch if error
       move.1    d0,ifd       save input file descriptor

* strip control characters from this file

main4  bsr.s     strip        subroutine to strip CTRLs
       bra.s     main2        look for more files

* finished all input files, terminate task

done   move.1    #0,d0        show normal termination
       sys       term



***********************************************


* subroutine to strip meaningless control characters
* from the file specified by file descriptor in "ifd*
* and write result to standard output.



*begin by reading a buffer full

strip   move.1   ifd,d0      get input file descriptor
        sys      read, buffer,BUFSIZ    read buffer full
        bes.s    rderr       branch if read error

* check for end of file (0 characters read)

       tst.1     d0          end of input file?
       beq.s     strip9      exit if so

* do actual stripping of control characters.  This will
* be done in place in the buffer by collapsing the data
* as meaningless control characters are stripped.

       move.1    #buffer,a0   point to source buffer
       move.1    a0,a1        point a1 to destination buffer
       bra.s     strip6       enter DBcc loop
strip4 move.b    (a0)+,d1     get a character into d1
       cmp.b     $#1F,d1      a control character?
       bhi.s     strip5       go keep character if not
       cmp.b     #$0D,d1      a carriage return
       beq.s    strip5       keep if so
       cmp.b    #$09,d1      a tab?
       bne.s    strip6       if not, don't keep
strip5 move.b   d1,(a1)+     put char. in buffer
strip6 dbra     d0,strip4    decrement count; loop if more

* finished stripping, a1 points to end of buffer of
* stripped data ready to be written

       sub.l    #buffer,a1   find no. of chars to write
       move.l   a1,wrtcnt    store in parameters
       move.l   #1,d0        write to standard output
       sys      ind,iwrite   do indirect write
       bes.s    writerr      branch if error
       bra.s    strip        go read another section

strip9 rts                   exit routine





********************************************

* error handling routines

opnerr  move.l   d0,-(a7)    save error status on stack
        move.l   #2,d0       standard error output
        sys      write,opners,opnerl
        bra.s    err
rderr   move.l   d0,-(a7)    save error status on stack
        move.l   #2,d0       standard error output
        sys      write,rderrs,rderrl
        bra.s    err
wrterr  move.l   d0,-(a7)    save error status on stack
        move.l   #2,d0       standard error output
        sys      write,wrterr,wrterl

err     move.l   (a7)+,d0    pull error status from stack
        sys      term        exit program


opners  fcc      "Can't open input file.",$d,0
opnerl  equ      *-opners
rderrs  fcc      'Error reading input file.',$d,0
rderrl  equ      *-rderrs
wrters  fcc      'Error writing output file.',$d,0
wrterl  equ      *-wrters


```

```
****************************************

* temporary storage and buffers

        data                   begin data segment

* indirect open system call parameters
iopen   dc.w     open          open function code
opname  dc.l     0             name of file to open
opmode  dc.l     0             open mode 1 (reading)

* indirect write system call parameters
iwrite  dc.w     write         write function code
wrtbuf  dc.l     buffer        buffer to write from
wrtcnt  dc.l     0             byte count to write

ifd     ds.l     1             input file descriptor

        ds.b     512-24        reserve up to 512-byte boundary
buffer  equ      *             start on 512-byte boundary
BUFSIZ  equ      $1000-512     multiple of 512 bytes
        ds.b     BUFSIZ        reserve space for buffer


        end      start
```

Section 3
THE ASSEMBLER

INTRODUCTION

The 4400 assembler supports conditional assembly as well as numerous other directives for convenient assembler control. The assembler executes in two passes and can accept any size file so long as sufficient memory is installed to contain the symbol table. Output from the assembler is in the form of a relocatable object file.

This section describes the operation and use of the Assembler and Linking Loader. The Assembler accepts most of the Motorola standard mnemonics for instructions, and fully supports the MC68000/68010/68020 instruction set. This section describes differences between the Motorola standard for instructions and those supported by the assembler.

This section is not intended to teach the reader assembly language programming nor the full details of the MC68000 instruction set. It assumes the user has a working knowledge of assembly language programming and a manual describing the MC68000 instruction set and addressing modes in full.

Throughout this section, angle brackets (< and >) are often used to enclose the description of a particular item. The angle brackets show that it is a single item even though the description may require several words. In addition, square brackets ([and]) are used to enclose an optional item.

Details of the instruction set, assembler syntax, and addressing modes were obtained from *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, Copyright 1984 by Motorola Incorporated.

Invoking the Assembler

Assembler text files must be standard text files with no line numbers or control characters (except for carriage returns and tabs). Once you have both the assembler and the edited source file on a disk or diskettes which are inserted in a powered-up system, you are ready to begin.

The Command Line

The minimum command line necessary to assemble a source file is:

```
++ asm sourcefile
```

When parameters are omitted, the assembler assumes default parameters. Two types of output are available from the assembler: object code and assembled source listing. (The options regarding the assembled source listing output is described a little later.) Object code is written into a operating system file. It is also possible to disable production of the object code file. Since no specifications are made concerning object code output in the above example, the assembler assumes the default case, which is to produce an object file. Since an output-file object is not specified, the input source file name is used with the characters *.r* appended. If there is not room to append those two characters, the last one or two characters of the input file name is truncated to make room. In our above example, the created binary file would be named *sourcefile.r*. Should a file exist with the same name, it will be automatically deleted with no prompting.

If you wish to create an object file with another name, place the desired file name on the command line as follows:

```
        ++ asm sourcefile +o=objectfile
```

The *+o=* is an option to the assembler which specifies that an object file is being created with the specified name. This example produces an object file named *objectfile*. Again, if a file by that name already existed, it would be deleted to permit creation of the new object file.

Multiple Input Source Files

The 4400 assembler is capable of accepting more than one file as the source for assembly. If multiple input files are specified, they are read in the calling order and assembled together to produce a single output file. This permits the user to break source programs down into more convenient size source files that may then be assembled into one object file. As mentioned, the files are read sequentially in the calling order with the last line of source from the current file being followed immediately by the first line of the ensuing file. All *end* statements in the source are effectively ignored and the assembly is terminated when the last line of the last source file is read.

There are two ways to specify multiple input files to the assembler: by entering the name of each file and by a match list in a file specification. Entering each filename would look like this:

```
        ++ asm file1 file2 file3 file4
```

A match list in the file specification looks like this:

```
        ++ asm file[1-4]
```

In this example, the square brackets do not denote an optional item, but rather are the method of specifying a list of match characters. Both of the above examples produces the same result. Note that in these examples an object file is created by default and is called *file1.r* (the name is taken from the first input file). As before, we can also specify an object file name as follows:

```
        ++ asm file1 file2 file3 file4 +o=command
```

This results in an object file called *command.*

Specifying Assembly Options

Now we shall go one step further and add a set of single character option flags that may be set on the command line as follows:

```
        ++ asm sourcefile +options
```

The plus sign is required to separate the option(s) from the file specification(s). In this example, the word *options* following the plus sign represents a single character option flag or list of character option flags which either enable or disable a particular option or options. In all cases, they reverse the sense of the particular option from its default sense. Any number of options may be specified and they may be specified in any order. There may not be spaces within the option list.

Following is a list and description of the available options:

+b	Do not create a binary file on the disk, even if an binary file name is specified. This is useful when assembling a program to check for errors before the final program is completed or when obtaining a printed source listing.
+e	Suppress end summary information. At the end of the assembly, the assembler may report the size of the segments and the total count of errors, warnings and excessive jumps. Often the user does not wish to have any output generated at all; the +e option suppresses this summary information. If this is used without selecting the +l and +s options, then it is possible that no output listing will be generated. However, if there are any errors reported in the module, this summary information is not suppressed.
+f	Disables the auto-fielding feature of the assembler such that assembled output lines appear in the exact form as found in the input file.
+F	Enable debug or *fix* mode. There are two forms of line comments. One begins with an asterisk (*) the other with a semicolon (;), both in the first column of the source line. If the comment begins with a semicolon, the +F option instructs the assembler to ignore the semicolon and process the line as though the semicolon never existed. The asterisk in the first column of a source line always denotes a comment regardless of the state of this option.
+l	Produce the assembled listing output. If specified, the assembler outputs each line as it is assembled in the second pass, honoring the "lis" and "nol" options (see the "opt" directive). Those lines containing errors will always be printed, regardless of whether or not this option is specified.
+L	Produce a listing of the file during the first pass of the assembler. The assembler prints unformatted lines (exactly as read) to standard output.

+n	Enables the printing of decimal line numbers on each output line. These numbers are the consecutive number of the line as read by the assembler. Error lines are always output with the line number, regardless of the state of this option.
+s	Produce the symbol table output. If this option is specified, the assembler produces a sorted symbol table at the end of an assembly. Note that the "l" option will not produce the symbol table output, just the source listing. In the symbol table, global symbols are preceded by an "*," and other symbols by a blank.
+S	Limit each symbol to only eight characters internally. Normally, the user can define and use symbols that contain 63 unique characters. However, in some cases, it may be necessary to limit the uniqueness of the symbols to only eight characters.
+t	Produce object code for the MC68000 rather than the MC68010/68020. This option affects only the code generation for the *Move from CCR* instruction. Normally the assembler produces the MC68010/68020 version of this instruction. If this option is specified, the assembler produces the MC68000 *Move from SR* instruction (Privileged on the MC68010/68020), in its place.
+u	Set all undefined symbols as external. In some cases you may wish to assemble a module that has some undefined external symbols. The +u option treats all undefined references as external references. The +u option should not substitute for the good programming practice of listing all external symbols in the operand field of the *extern* directive.
+o=<filename>	*Filename* Allows specification of an output object file name (in this example *file*).

Order for Specifying Filenames, Options, and Parameters

Input filenames, options, and command line parameters can be specified to the assembler in any order. The assembler scans the input command line twice, once to pick out all options and parameters (they all begin with a plus sign) and then again to pick out all file specifications. Place order is significant only when multiple input files are specified. They are assembled in the order entered on the calling line.

Sending Output to a Hardcopy Device

The assembler uses the facilities of the 4400's operating system to send the assembled listing to a hardcopy device. The most common means are to route the standard output to a file that may later be printed.

Examples:

```
        ++ asm test
```

Assembles a file called *test* and creates an binary file called *test.r* in the same directory. No listing is output (except for any lines with errors) and no symbol table is output.

```
        ++ asm test +ls
```

Same as before except that assembled listing is output to the terminal, as is the symbol table.

```
        ++ asm test +o=/bin/test +ls
```

Assembles a file called *test* in the current directory and produces an object file in the *bin* directory called *test.* The listing and symbol table are output to the terminal, and if a file by the name of *test* already resides in the *bin* directory, it is automatically deleted before the assembly starts.

```
        ++ asm /john/main +bnl
```

This command assembles the file *main* in John's directory but does not produce a binary file. The assembled listing is output with line numbers. No symbol table is printed.

```
        ++ asm file[1-4] +bln
```

This command assembles all files beginning with *file* and ending with a 1, 2, 3, or 4. No binary or symbol table is output, and line numbers are turned on.

```
        ++ asm +u dumper +nel
```

This command demonstrates the fact that the filenames, and options can come in any desired order on the command line. The file to be assembled is called *dumper.* The assembled listing is output with line numbers. All undefined references are made external, no summary information will be output, and no symbol table is produced.

ASSEMBLER OPERATION & SOURCE LINE COMPONENTS

The 4400 assembler is a two-pass assembler. In Pass One a symbolic reference table is constructed and, in Pass Two the code is actually assembled, and a listing and object code are produced if requested. The source may be supplied in free format, as described below. Each source line consists of the actual source statement, terminated with a carriage return (0D hex). The source must be comprised of ASCII characters with their parity or 8th bit cleared to zero. Special meaning is attached to many of these characters as will be described later. Control characters ($00 to $FF) other than the carriage return ($0D) and horizontal tab ($09) should not be in the actual source statement part of the line. Their inclusion in the source statement produces undefined results.

Each source line consists of up to four fields: Label, Opcode, Operand, and Comment. With two exceptions, every line must have an opcode while the other fields may or may not be optional. These two exceptions are:

1. *Comment Lines* can be inserted anywhere in the source and are ignored by the assembler during object code production. Comment lines can be either of two types:
   a. Any line beginning with an asterisk (hex 2A) or semicolon (hex 3B) in column one.
   b. A null line or a line containing only a carriage return. While this line can contain no text, it is still considered a comment line as it causes a space in the output listing.
2. Lines which contain a label but no opcode or operand field.

Source Statement Fields

The following pages describe the four source statement fields and their format specifications. The fields are free format which means there can be any number of spaces separating each field. In general, no spaces are allowed within a field.

Label or Symbol Field

This field may contain a symbolic label or name that is assigned the instruction's address and may be called upon throughout the source program.

1. Ordinary Labels
   a. The label begins in column 1 and must be unique. Labels are optional. If the label is omitted, the first character of the line must be a space.
   b. A label may consist of letters (A-Z or a-z), numbers (0-9), or an underscore (_ or 5F hex). Note that upper and lower case letters are not considered equivalent. Thus *ABC* is a different label from *Abc*.
   c. Every label must begin with a letter or underscore.
   d. Labels can be of any length, but only the first 63 characters are significant.
   e. The label field must be terminated by a space, tab, or a return.
2. Local Labels
   a. Local labels follow many of the same rules as ordinary labels. They begin in column one and they must be terminated by a space, tab or return.
   b. Local labels consist of a number from 0 to 99. These numbers may be repeated as often as desired in the same source module; they need not be in numerical order. Note that the labels *00* and *0*, *01* and *1*, etc., are unique labels.
   c. Local labels may be treated as ordinary labels; however, they cannot be global or external. They can not be used in the label field of an *equ* or *set* directive.
   d. Local labels are referenced by using the local label number terminated with an "f" for first forward reference found or a "b" for the first backward reference found. A backward or forward reference can never refer to the same line that it is found on. For example,

```
1   beq 2f   "2f" => next occurrence of "2"
2   jsr xx   both branches point here
3   bra 2b   "2b" => previous occurrence of "2"
```

   e. Local labels should be used primarily (but not necessarily exclusively) for branching or jumping around some sections of code. In most cases, branching around a few lines of code does not warrant the use of an ordinary label. When making a reference to a nearby location in the program there is often no appropriate name with much significance; therefore, programmers have tended to use symbols like 11,12, etc. This can lead to the danger of using the same label twice. Local labels have freed the programmer from the necessity of thinking of a symbolic name of a location. Furthermore, local labels require less storage internally and lookup is faster than with ordinary labels. A maximum of 500 local labels can be used in one module.

Opcode Field

This field contains the opcode (mnemonic) or a pseudo-op. It specifies the operation to be performed. The pseudo-ops recognized by this assembler are described later in this section.

1. The opcode is made up of letters (A-Z or a-z). In this field, upper and lower case can be used interchangeably.
2. This field must be terminated by a space or tab if there is an operand or by a space, tab, or return if there is no operand.
3. The opcode may have a length specification associated with it. This length specification indicates whether the operation is to take place on bytes, words, or long words. The default is words. The specification consists of a period followed by one of the letters *b*, *w*, *l*, or *s*. Upper case letters are also permitted. The following summarizes the specifications:

.b or .B	bytes (8-bits)
.w or .W	words (16-bits, the default)
.l or .L	long words (32-bits)
.s or .S	short specification (for branches)

Operand Field

The operand provides data or address information required by the opcode. This field may or may not be required, depending on the opcode. Operands are generally combinations of register specifications and mathematical expressions. See the heading of *Expressions*, later in this section for the rules for forming valid expressions.

1. The operand field can contain no spaces or tabs.
2. This field is terminated with a space, tab, or return.
3. Any of several types of data may make up the operand: register specifications, numeric constants, symbols, ASCII literals.

Comment Field

The comment field may be used to insert comments on each line of source. Comments are for the programmer´s convenience only and are ignored by the assembler.

1. The comment field is always optional.
2. This field must be preceded by a space or tab.
3. Comments may contain any characters from SPACE (hex 20) through DELETE (hex 7F) and the tab character.
4. This field is terminated by a carriage return.

Register Specification

Many opcodes require that the operand following them specify one or more registers. Both lower and upper case are allowed. The following are possible register names:

D0-D7	Data Registers
A0-A7	Address Registers
A7, SP	System stack pointer of the active system state
USP	User stack pointer
CCR	Condition Code Register (Part of SR)
SR	Status Register
VBR	Vector Base Register (MC68010/20)
SFC	Source Function Code Register (MC68010/20)
DFC	Destination Function Code Register (MC68010/20)
CAAR	Cache Address Register (MC68020)
CACR	Cache Control Register (MC68020)
MSP	Master Stack Pointer (MC68020)
ISP	Interrupt Stack Pointer (MC68020)

Expressions

Many operands must include an expression. This expression may be one or more items combined by any of four operator types: arithmetic, logical, relational, and shift.

Expressions are always evaluated as full 32-bit operations. If the result of the operation is to be fewer bits, the assembler truncates the upper part.

An expression must not contain any embedded spaces or tabs.

Item Types

The item or items in an expression may be any of the four types listed below. These may stand alone or may be intermixed by the use of the operators.

1. NUMERICAL CONSTANTS: Numbers may be supplied to the assembler in any of the four number bases shown below. The number given is converted to 32 bits truncating any numbers greater than that. If smaller numbers are required, the 32-bit number is then further truncated to the proper size. To specify which number base is desired, the programmer must supply a prefix character to a number.

BASE	PREFIX	CHARACTERS ALLOWED
Decimal	none	0 thru 9
Binary	%	0 or 1
Octal	@	0 thru 7
Hexadecimal	$	0 thru 9, A thru F

   If no prefix is assigned, the assembler assumes the number to be decimal.

2. ASCII CONSTANTS: ASCII constants are specified in expressions by enclosing the string in single or double quotation marks. The string must consist of one to four characters, depending on the desired size attribute. The specified characters may not include control characters (must be between 20 hex and 7F hex inclusive).

3. LABELS: An expression may contain labels which have been assigned some address, constant, relocatable or external value. As described above under the label field, a label consists of letters, digits, and underscores beginning with a letter or underscore. The label may be of any length, but only the first 63 characters are significant. Any label used in the operand field must be defined elsewhere in the program. Local labels may also be used in the operand field. None of the standard register specifications should be used as a label.

4. PC DESIGNATOR: The asterisk (*) has been set aside as a special PC designator (Program Counter). It may be used in an expression just as any other value and is equal to the address of the current instruction. The value of the PC designator is relocatable in the text, data or bss segments; its value is given at load time.

Types of Expressions

Three types of expressions are possible in the 4400 assembler: absolute, relocatable and external expressions.

Absolute Expressions

An expression is absolute if its value is unaffected by program relocation. An expression can be absolute, even though it contains relocatable symbols, under both of the following conditions:

1. The expression contains an even number of relocatable elements.
2. The relocatable elements must cancel each other. That is, each relocatable element (or multiple) in a segment must be canceled by another element (or multiple) in the same segment. In other words, pairs of elements in the same segment must have signs that oppose each other. The elements that form a pair need not be contiguous in the expression.

For example, text1 and text2 are two relocatable symbols in the text segment; the following examples are absolute expressions.

```
text1-text2
5*(text1-text2)
```

Relocatable Expressions

An expression is relocatable if its value is affected by program relocation in a relocatable module. A relocatable expression consists of a single relocatable symbol or, under all three of the following conditions, a combination of relocatable and absolute elements.

1. The expression does not contain an even number of relocatable elements.
2. All the relocatable elements but one must be organized in pairs that cancel each other. That is, for all but one segment, each relocatable element (or multiple) in a segment must be canceled by another element (or multiple) in the same block.
3. The uncancelled element can have either positive or negative relocation.

For example, text1 and text2 are symbols from the text segment, data1 and data2 are symbols from the data segment, and bss1 and bss2 are symbols from the bss segment; the following examples are relocatable:

-bss2+3*5+(data2-data2)	negative relocation from bss segment
text1+(data1-data2)+(bss2-bss1)	relocation from text segment
data1-(bss2-bss1)	relocation from data segment
*	(PC Designator) relocation from current segment

External Expressions

An expression is external if its value depends upon the value of a symbol defined outside of the current source module. An external expression can consist of a single external symbol, or, under both of the following conditions, an external expression may consist of an external symbol, relocatable elements and absolute elements:

1. The expression contains an even number of relocatable elements
2. The relocatable elements must cancel each other. That is, each relocatable element (or multiple) in a segment must be canceled by another element in the same segment. In other words, pairs of elements in the same segment must have signs that oppose each other.

For example, if ext1 is an external symbol, text1, text2, data1, data2, bss1, bss2 all have the same meaning as above in the previous examples; then the following examples are external:

```
(text1-text2)+ext1-(data2-data1)
5+ext1-3
3/(text2-text1)-ext1
```

Expression Operators

Operators permit operations such as addition or division to take place during the assembly, and the result becomes a permanent part of your program. Many of these operators will only apply to absolute symbols and expressions. It does not make sense to multiply a relocatable or external value at assembly-time! Only the + and - operators can apply to relocatable and external symbols and expressions.

Arithmetic Operators

The arithmetic operators are:

Operator	Meaning
+	Unary or binary addition
-	Unary or binary subtraction
*	Multiplication
/	Division (any remainder is discarded)

Logical Operators

The logical operators are:

Operator	Meaning
&	Logical AND operator
|	Logical OR operator
!	Logical NOT operator
>>	Shift right operator
<<	Shift left operator

The logical operations are full 32-bit operations. In other words for the AND operation, every bit from the first operand or item is individually ANDed with its corresponding bit from the second operand or item. The shift operators shift the left term the number of places indicated by the right term. Zeroes are shifted in and any bits shifted out are lost.

Relational Operators

The relational operators are:

Operator	Meaning
=	Equal
<	Less than
>	Greater than
<>	Not equal
<=	Less than or equal
>=	Greater than or equal

The relational operations yield a true-false result. If the evaluation of the relation is true, the resulting value be all ones. If false, the resulting value is all zeros. Relational operations are generally used in conjunction with conditional assembly, as shown in that discussion.

Operator Precedence

Certain operators take precedence over others in an expression. This precedence can be overcome by the use of parentheses. If there is more than one operator of the same precedence level, and no parentheses indicate the order in which they should be evaluated, then the operations are carried out in left to right order.

The following list classifies the operators in order of precedence (highest priority first):

1. Parenthesized expressions
2. Unary + and -
3. Shift operators
4. Multiply and Divide
5. Binary + and -
6. Relational Operators
7. Logical NOT Operator
8. Logical AND and OR Operators

INSTRUCTION SET DIFFERENCES

This discussion describes the differences in the instruction mnemonics accepted by the assembler and the Motorola *standard.* The standard is assumed to be that defined in the *MC68000 16-Bit Microprocessor User's Guide,* published by Motorola Semiconductor Products, Inc. It is assumed that the reader is familiar with the contents of the *Instruction Set Details* portion of that manual. In particular, the user should be familiar with the description of the assembler syntax that accompanies the discussion of the individual instructions.

The assembler recognizes the standard instruction set with the exception of some of the so-called *variations.* Having a specific opcode for these variations is not necessary, because the assembler can infer their existence from an analysis of the operands and generate the proper code. This relieves the programmer from the need for remembering the opcodes, and the particulars of each. The variations that are handled in this manner are: address, quick, and immediate. Note that the *extend* variation is still supported. Thus, the following instructions are not specifically recognized by the Assembler:

ADDA, ADDQ, ADDI	Use ADD instead
ANDI	Use AND instead
CMPA, CMPI, CMPM	Use CMP instead
EORI	Use EOR instead
MOVEA, MOVEQ	Use MOVE instead
ORI	Use OR instead
SUBA, SUBQ, SUBI	Use SUB instead

Remember that even though these mnemonics are not recognized, the assembler can and does generate code for address, quick, and immediate instructions. The proper instruction is selected automatically after analyzing the operands.

Instruction Set Extensions

The following instruction extensions are recogized by the assembler and are valid only with the 68020 processor. If you use these extensions and attempt to generate compiled code for a 68000 or 68010 microprocessor, the assembler gives you one of the following error messages:

```
*** Error - Unknown instruction.

*** Error - Unknown addressing mode.
```

Mnemonic	Description
Bcc	Supports 32-Bit Displacements
BFxxxx	Bit Field Instructions (BFCHG, BFCLR, BFEXTS, BFEXTU, BFEXTS, BFFFO, BFINS, BFSET, BFTST)
BKPT	New Instruction Functionality
BRA	Supports 32-Bit Displacement
BSR	Supports 32-Bit Displacement
CALLM	New Instruction
CAS,CAS2	New Instruction
CHK	Supports 32-Bit Operands
CHK2	New Instruction
CMPI	Supports Program Counter Relative Addressing Modes
CMP2	New Instruction
cp	Coprocessor Instructions
DIVS/DIVU	Supports 32-Bit and 64-Bit Operands
EXTB	Supports 8-Bit Extend to 32-Bits
LINK	Supports 32-Bit Displacement
MOVEC	Supprts New Control Registers
MULS/MULU	Supports 32-Bit Operands
PACK	New Instruction
RTM	New Instruction
TST	Supports Program Counter Relative Addressing Modes
TRAPcc	New Instruction
UNPK	New Instruction

The default data size is *word*. Instructions that can manipulate more than one size of data item may be modified by postfixing a data length specification to the opcode. The data length specifications are:

.l or .L	For long word (32 bits)
.w or .W	For word (16 bits, the default)
.b or .B	For byte (8 bits)

Addressing Modes

For information about the addressing modes of the processors in the 4400 series family of products refer to:

Title	Motorola Part Number
M68000 Programmer´s Reference Manual	M68000UM(AD4)
MC68020 32-Bit Microprocessor User´s Manual	MC68020UM/AD
MC68881 Floating-Point Coprocessor User´s Manual	MC68881UM/AD

Convenience Mnemonics

CLC	Clear carry condition code bit
CLN	Clear negative condition code bit
CLV	Clear overflow condition code bit
CLX	Clear extend condition code bit
CLZ	Clear zero condition code bit
SEC	Set carry condition code bit
SEN	Set negative condition code bit
SEV	Set overflow condition cod
SEX	Set extend condition code bit
SEZ	Set zero condition code bit

STANDARD DIRECTIVES OR PSEUDO-OPS

Besides the standard machine language mnemonics, the assembler supports several directives or pseudo-ops. These are instructions for the assembler to perform certain operations, and are not directly assembled into code. There are three types of directives in this assembler: those associated with conditional assembly, those associated with macros, and those which generally can be used anywhere which we shall call *standard directives*. The standard directives are:

dc	fcc	opt	spc
ds	fdb	pag	sttl
equ	fqb	rab	sys
err	info	rmb	ttl
even	lib	rzb	
fcb	log	set	

Other types of directives are explained in other sections, but are listed here for completeness:

Conditional Directives	Relocation Directives	
if	base	end
ifn	bss	extern
else	common	global
endif	endcom	name
	data	struct
	define	text
	enddef	

dc

The *dc* or Define Constant directive defines one or more constants in memory. A size specification may be postfixed to the directive to indicate that the constant is to be stored in bytes, words, or long words. The default is *words*. If multiple operands are specified, the effect is as though the operands appeared in consecutive *dc* directives. The operands may be actual values (constants or ASCII strings) or expressions. ASCII strings must be enclosed in single quotation marks.

The constant is aligned on the proper boundary, depending on the size specification (byte boundary for *.b*, word boundary for *.w*, and long word boundary for *.l*). When ASCII strings are specified with a word or long word size specification, the string will be padded on the right with zero bytes if there are not enough characters to exactly fill the last word or long word. If an ASCII string is specified with a byte size specification, and the instruction or directive following the *dc.b* directive requires word or long word alignment, then zeroes are appended to the character string to force such alignment. Some examples:

label1 dc.b 3,7,´String´	
label2 dc.w 123,´abc´,98	The ´abc´ will be padded with a zero byte
dc.l ´a´,131072	The ´a´ will be padded with 3 zero bytes

ds

The *ds* or Define Storage directive reserves areas of memory. The reserved memory is not guaranteed to be initialized in any way. A size specification may be postfixed to the directive to indicate that bytes, words, or long words are to be reserved. If words or long words are specified, the reserved memory is properly aligned. A single operand indicates how many bytes, words, or long words are to be reserved. If a label is present, its value is the address of the lowest memory location reserved. If the value of the operand is zero, no space is reserved; however, alignment takes place if *ds.w* or *ds.l* is specified. Some examples:

ds.b 20	reserve 20 bytes
ds 10	reserve 10 words
ds.l 5	reserve 5 long words
ds.l 0	force alignment on long word boundary

equ

The *equ* or Equate directive equates a symbol to the expression given in the operand. No code is generated by this statement. Once a symbol is equated to some value, it can not be changed at a later time in the assembly. The form of an equate statement is

<label> equ <nonexternal expression>

The label is strictly required in equate statements. Absolute or relocatable expressions are allowed; external expressions are illegal. If the expression is relocatable, both the value and the attribute is assigned to the label.

err

The *err* directive may be used to insert user-defined error messages in the output listing. The error count is also incremented by one. The format is:

err <message to be printed>

All text past the *err* directive (excluding leading spaces) is printed as an error message (preceded by three asterisks) in the output listing. Note that the *err* directive line itself is not printed. A common use for the *err* directive is in conjunction with conditional assembly, to report user-defined illegal conditions.

even

The *even* directive is used to force the program counter to an even address (word boundary).

fcb

The *fcb* or Form Constant Byte directive is used to set associated memory bytes to some value as determined by the operand. *Fcb* may be used to set any number of bytes, as shown below:

[<label>] fcb <expr. 1>,<expr. 2>, . . . ,<expr. n>

<expr. x> stands for some absolute, relocatable or external expression. Each expression given (separated by commas) is evaluated to 8 bits, and the resulting quantities are stored in successive memory locations. The label is optional.

fcc

The *fcc* or Form Constant Character directive allows the programmer to specify a string of ASCII characters delimited by some non-alphanumeric character such as a single quote. All the characters in the string is converted to their respective ASCII values and stored in memory, one byte per character. Some examples:

```
label1  fcc  'This is an fcc string'
label2  fcc  .so is this.
        fcc  /Labels are not required./
```

There is another method of using *fcc* which is a deviation from the standard Motorola definition of this directive. This method allows you to place certain expressions on the same line as the standard *fcc* delimited string. The items are separated by commas and are evaluated to 8-bit results. In some respects this is like the *fcb* directive. The difference is that in the *fcc* directive, expressions must begin with a letter, number or dollar sign, whereas in the *fcb* directive any valid expression will work. For example, %10101111 is a valid expression for a *fcb* but not for a *fcc* since the percent-sign would look like a delimiter and the assembler would attempt to produce 8 bytes of data from 8 ASCII characters which follow (a *fcc* string). The dollar sign is an exception to allow hex values such as $0D (carriage return) to be inserted along with strings. Some examples:

```
intro   fcc   'This string has CR & LF',$D,$A
        fcc   'string 1',0,'string 2'
        fcc   $04,extlabel,/delimited string/
```

Note that more than one delimited string may be placed on a line as in the second example.

fdb

The *fdb* or Form Double Byte directive is used to create 16 bit constants in memory. It is exactly like the *fcb* directive except that 16 bit quantities are evaluated and stored in memory for each expression given. The form of the statement is:

[<label>] fdb <expr. 1>,<expr. 2>,. . . ,<expr. n>

Again, the label field is optional. The generated data is guaranteed to be on a word boundary (see the *dc* directive).

fqb

The *fqb* or Form Quad Byte directive is used to create 32-bit constants in memory. It is exactly like the *fdb* directive, except that 32-bit quantities are evaluated and stored in memory for each expression given. The form of the statement is:

[<label>] fdb <expr. 1>,<expr. 2>,. . . ,<expr. n>

Again, the label field is optional. The generated data is guaranteed to be on a word boundary (see the *dc* directive).

info

The *info* directive allows the user to store textual comments in a binary file. A 4400 user can execute the command *info* and view the text on the screen. The assembler's *info* directive places all text following the *info* command (excluding leading spaces) into a temporary file called */tmp/asmbinfoxxxxx*, where xxxxx represents the current task number. At the end of the assembly, all text stored in this temporary file is appropriately copied into the normal binary file, and the temporary file is then deleted. Syntax is as follows:

```
info This is a comment for the binary file.

info It is a convenient way of inserting version nos.

info Version X.XX - Released XX/XX/XX
```

Any number of *info* directives may be inserted at any point in the source listing. No label is allowed, and no actual binary code is produced.

lib

The *lib* or Library directive allows the user to specify an external file for inclusion in the assembled source output. Under normal conditions, the assembler reads all input from the file(s) specified on the calling line. The *lib* directive allows the user to temporarily obtain the source lines from some other file. When all the lines in that external file have been read and assembled, the assembler resumes reading of the original source file. The proper syntax is:

```
lib <file spec>
```

where <file spec> is a standard 4400 file specification.

The assembler first looks for the specified file in the current directory. If the file isn't found in the current directory, the assembler then looks for a directory named *lib* in the current directory. If it finds such a directory, the assembler attempts to find the specified file in that *lib* directory. If not found there, the assembler makes a third and final attempt to find the specified file by looking in the directory */lib*. If the file is not found in any of these three directories, the assembler reports an error.

Any *end* statements found in the file called by the *lib* directive are ignored. The *lib* directive line itself does not appear in the output listing. Any number of *lib* instructions may appear in a source listing. It is also possible to nest *lib* files up to 7 levels.

log

The *log* directive is used to calculate the log, base 2, of an absolute expression. The result is 32 bits. The statement acts like a *set* statement, in that the label specified can be redefined with other *log* directives or *set* directives. The form of the statement is:

```
<label> log <absolute expression>
```

The label field is strictly required.

opt

The *opt* or Option directive allows the user to choose from several different assembly options. These options are generally related to the format of the output listing and object code. The options that can be set with this command are listed below. The proper form of this instruction is:

```
opt <option 1>,<option 2>,. . . ,<option n>
```

Note that any number of options canbe given on one line if separated by commas. No label is allowed, and no spaces or tabs may be embedded in the option list. The options are set during Pass Two. If contradicting options are specified, the last one on the command line takes precedence. If a particular option is not specified, the default case for that option takes effect. The default cases are signified below by an asterisk.

The allowable options are:

con	print conditionally skipped code
noc*	suppress conditional code printing
lis*	print an assembled listing
nol	suppress output of assembled listing

The *lis* and *nol* options can be used to selectively turn parts of a program listing on or off as desired. If the *+l* command line option is specified, however, the *lis* and *nol* options are overridden and no listing occurs.

pag

The *pag* directive causes a page eject in the output listing and prints a header at the top of the new page. Note that the *pag* option must be enabled in order for this directive to take effect. It is possible to assign a new number to the new page by specifying such in the operand field. If no page number is specified, the next consecutive number is used. No label is allowed and no code is produced. The *pag* operator itself does not appear in the listing unless some sort of error is encountered. The proper form is:

```
pag [<expression>]
```

The expression is optional. The first page of a listing does not include the header and is considered to be page 0. Thus, all options, title, and subtitle may be set up and followed by a *pag* directive to start the assembled listing at the top of page 1 without the option, title, or subtitle instructions being in the way.

rab

The *rab* or Reserve Aligned Bytes directive is used to reserve areas of memory for data storage. The bytes are forced to a word boundary. The number of bytes specified by the expression in the operand are skipped during assembly. No code is produced in those memory location and therefore the contents are undefined at run time. The proper usage is shown here:

```
[<label>] rab <absolute expression>
```

The label is optional, and the absolute expression is a 32-bit quantity. *Rab* directives found in the text or data segments act like *rzb*, and produce code which is guaranteed to be on an even boundary.

rmb

The *rmb* or Reserve Memory Bytes directive reserves areas of memory for data storage. The number of bytes specified by the expression in the operand are skipped during assembly. No code is produced in those memory locations and therefore the contents are undefined at run time. The proper usage is:

```
[<label>] rmb <absolute expression>
```

The label is optional, and the absolute expression is a 32-bit quantity. Any *rmb* directives found in the text or data segments act like *rzb*, and produce code.

rzb

The *rzb* or Reserve Zeroed Bytes directive is used to initialize an area of memory with zeroes. Beginning with the current PC location, the number of bytes specified is set to zero. The proper syntax is:

```
    [<label>] rzb <absolute expression>
```

where the absolute expression is a 32-bit expression. This directive does produce object code. Any *rzb* directives found in the bss segment act like *rmb*.

set

The *set* directive sets a symbol to the value of some expression, much as an *equ* directive. The difference is that a symbol may be *set* several times within the source (to different values), but may be *equated* only once. If a symbol is *set* to several values within the source, the current value of the symbol will be the value last *set*. The statement form is:

```
    <label> set <nonexternal expression>
```

The label is strictly required, and no code is generated.

spc

The *spc* or Space directive inserts the specified number of spaces (line feeds) into the output listing. The general form is:

```
    spc [<space count>[,<keep count>]]
```

The space count can be any number from 0 to 255. If the page option is selected, *spc* does not cause spacing past the top of a new page. The <keep count>, which is optional, is the number of lines to keep together on a page. If there are not enough lines left on the current page, a page eject is performed. If there are <keep count> lines left on the page (after printing <space count> spaces), output continues on the current page. If the page option is not selected, the <keep count> is ignored. If no operand is given, the assembler defaults to one blank line in the output listing.

sttl

The *sttl* or Subtitle directive is used to specify a subtitle to be printed just below the header at the top of an output listing page. It is specified much as the *ttl* directive:

sttl <text for the subtitle>

The subtitle may be up to 52 characters in length. If the page option is not selected, this directive is ignored. As with the *ttl* option, any number of *sttl* directives may appear in a source program. The subtitle can be disabled or turned off by an *sttl* command with no text following.

sys

The *sys* or system call directive allows the programmer to setup a system call. Such a call consists of a TRAP#15 instruction followed by a two byte function code optionally followed by 32-bit parameter values. This directive automatically inserts the TRAP, then obtains the function code and any other parameters from the operand field.

sys <function>,<parameter1>,<parameter2>, . . .

The <function> and <parameter> values may be any legal absolute, relocatable or external expression. <function> will be stored as 16 bits, all <parameters> will be stored as 32-bits.

ttl

The *ttl* directive allows the user to specify a title or name to the program being assembled. If the *pag* option is also selected, this title is then printed in the header at the top of each output listing page. If the page option is not selected, this directive is ignored. The proper form is:

ttl <text for the title>

All the text following the *ttl* directive (excluding leading spaces) is placed in the title buffer. Up to 32 characters are allowed, with any excess being ignored. It is possible to have any number of *ttl* directives in a source The latest one encountered will always be the one used for printing at the top of the following page(s).

CONDITIONAL ASSEMBLY

The assembler supports conditional assembly — the ability to assemble only certain portions of your source program depending on the conditions at assembly time. Conditional assembly is particularly useful in situations where you might need several versions of a program with only slight changes between versions.

As an example, suppose you required a different version of some program for four different systems whose output routines varied. Rather than prepare four different source files, you could prepare one that would assemble a different set of output routines depending on some variable that was set with an *equ* directive near the beginning of the source. Then it would only be necessary to change that one *equ* statement to produce any of the four final programs.

The *if-endif* Clause

In its simplest form, conditional assembly is performed with two directives: *if* and *endif*. The two directives are placed in the source listing in that order with any number of lines of source between. The assembler evaluates the expression associated with the *if* statement (we will discuss this expression in a moment), and if the result is true, assembles all the lines between the *if* and *endif* and then continues assembling the lines after the *endif*. If the result of the expression is false, the assembler will skip all lines between the *if* and *endif* and resume assembly of the lines after the *endif*. The syntax of these directives is:

```
if <expression>
 .
 . conditional code goes here
 .
endif
```

The *endif* directive requires no additional information, but the *if* directive requires an expression. This expression is considered FALSE if the 32-bit result is equal to zero. If the result is not equal to zero, the expression is considered TRUE.

The *if-else-endif* Construction

An *else* directive may be placed between the *if* and *endif* statements. In effect, the lines of source between the *if* and *endif* are split into two groups by the *else* statement. Those lines before the *else* are assembled if the expression is true; those after (up to the *endif*) are ignored. If the expression is false, the lines before the *else* are ignored while those after it are assembled. The *if-else-endif* construct appears as:

```
if <expression>
.
. this code is assembled if the expression is true
.
else
.
. this code is assembled if the expression is false
.
endif
```

The *else* statement does not require an operand. There can be only one *else* between an *if-endif* pair.

It is possible to nest *if-endif* clauses (including *else*s). That is, an *if-endif* clause may be part of the lines of source found inside another *if-endif* clause. You must be careful, however, to terminate the inner clause before the outer.

Another form of the conditional directive, *ifn* (*if not*) functions just like *if*, except that the sense of the test is reversed. Thus, the code immediately following is assembled if the result of the expression is NOT TRUE. An *ifn-else-endif* clause appears as follows:

```
ifn <expression>
.
. this code is assembled if the expression is FALSE
.
else
.
. this code is assembled if the expression is TRUE
.
endif
```

NOTE

For conditionals to function properly, they must evaluate to the same result in Pass One and Pass Two. Thus, if labels are used in a conditional expression, they must be defined in the source before the conditional directive is encountered.

SPECIAL FEATURES

End of Assembly Information

Upon termination of an assembly and before the symbol table is output, three items of information may be printed: the total number of errors encountered, the total number of excessive branches encountered, and the sizes of the text, data and bss segments.

The number of errors is printed in the following manner:

```
0 Errors detected.
```

Excessive branches (a long branch used where a short branch will suffice) are printed after the error count, for example:

```
1 Error detected.
3 Excessive branches detected.
```

The size of the segments are displayed as follows:

```
SEGMENT SIZES
TEXT SEGMENT = 00002C
DATA SEGMENT = 00010A
BSS SEGMENT  = 000006
```

All of this information may be suppressed by using the "+e" command line option; however, if errors are detected, this information is displayed anyway.

Excessive Branch Indicator

To allow size and speed optimization of the final code, the assembler places a greater-than sign just before the address of any long branch instruction that can be replaced by a short branch. The total count is reflected in the end-of-assembly information previously described. The following section of code shows just how it looks:

```
000000                          text
000000 4A80                     tst.l   d0
000002 6600 0006                bne     lab1
000006 4A81                     tst.l   d1
000008 6602                     bne.s   lab2
00000A 2601             lab1    move.l  d1,d3
00000C 2800             lab2    move.l  d0,d4
00000E                          end
```

Note how the *.s* postfix was used to create a short branch.

Auto Fielding

The output assembly listing automatically places the four fields of a source line (label, mnemonic, operand, and comment) in columns. This allows the programmer to edit a condensed source file without impairing the readability of the assembled listing. The common method of doing this is to separate the fields by only one space when editing. The assembled output places all labels in column 25, all opcodes in column 34, and all operands in column 42 and comments start in column 56 unless the operands field extends into the comments. There are a few cases where this automatic fielding can break down (such as lines with errors), but these cases are rare and generally cause no problem. Labels that are longer than 8 characters are printed on a line by themselves (above the code they are with — if any).

Fix Mode

Comment lines may begin with either an asterisk (*) or a semicolon (;). If a semicolon is used, the *+F* option of the assembler assumes that the *comment* is a valid instruction to be assembled. Therefore, the assembler acts as though the semicolon did not exist at all; the rest of the information on that line is assembled. For example:

```
;label1 move.l #2,d0
; sys term
```

With the *+F* option invoked, these two lines are assembled. This aides in the debugging process.

Local Labels

Local labels are available in the assembler. These local labels allow the programmer to reuse labels; in this way meaningless labels can be replaced with local labels. For more information on local labels, refer to the description of the label field in the *Assembler Operation and Source Line Components* discussion earlier in this section.

OBJECT CODE PRODUCTION

The object code output from the 4400 assembler is a standard 4400 relocating binary file for relocatable modules. This object code output can be turned on or off via the "+b" option on the calling line. The relocatable output module always requires processing by the linking loader to be executed. For more information about relocatable modules, refer to the discussion of the linking loader, later in this section.

Relocatable (Segmented) Object Code files

The 4400 operating system supports *segmentation* of binary files. It permits binary files to be broken into three segments of code: called *text*, *data*, and *bss*. Each assembly module must contain one of these directives before any instructions that produce object code can be processed. The assembler does not default to any given segment when assembling a file.

Any code in a *text* segment is assumed by the operating system to be read-only. That is, it will only read code in a *text* segment and will not attempt to write into it.

The *data* segment is sometimes referred to as *initialized data.* It is code that has been produced by the assembler and can be either read or written. For example, the data segment might contain a temporary variable that requires an initial value. At any point, the variable could be read or re-written.

The *bss* segment is an area of reserved memory where no actual code has been produced by the assembler. It is sometimes referred to as *uninitialized data.* The binary file does not contain any code to be placed in this section of memory, only a size value for this segment. Its main purpose is to tell the operating system that memory is required in this area, but it does not need to be initialized to any values. The *bss* segment or area of memory can be read or written.

Breaking the binary file into these three sections provides several benefits. The *text* segment is known to be read-only. This implies the code is never altered as long as the program runs. The operating system can make use of this fact by sharing this segment of memory in the event that more than one users wish to run the program at the same time. This can mean a considerable increase in efficiency of the system. The *data* section must be different for each user running the program. It is information (actual instructions or data) which must be initialized or loaded, but which can be altered at some later point. The *bss* segment really contains no code or data in the binary file. It is just a signal to the operating system that when the file is loaded it needs memory allocated in the area specified. The program should not assume that the memory in this segment is initialized to any particular value.

The assembler performs segmentation by maintaining three distinct location counters or program counters (PC´s). At any point in the assembly, only one of these PC´s is in effect. Any code generated by an instruction at that point is assembled at the address in the PC currently in effect. It is possible to switch to a different PC by use of one of the following three directives in the opcode field:

```
    text
    data
    bss
```

It is necessary to state the segment that is desired before any executable code is produced. It is possible to change which segment code is currently being generated into at any time. In other words, you could begin with a *text* directive, enter 10 lines of code, then switch to the data segment with a *data* directive, enter 10 lines of code, then switch back to the text segment with another *text* directive, etc. To resume with the last address used by a particular segment, enter the segment directive:

```
text
move.l    10,d0
data
temp      fcb       0
text
move.l    temp,a0
end
```

It is not possible to generate code in a *bss* segment. Any attempt to do so results in an error.

Code generated into the *data* segment is actually written to a temporary file called */tmp/asmbdataxxxxx* (xxxxx represents the current task number). At the end of the assembly, this data is copied onto the end of the text code found in the main output file, and the temporary file is immediately deleted.

The Base and Struct Directives

Two other directives related to PC´s and segmentation are *base* and *struct*. These directives are used almost exactly like a segment PC directive (especially the *bss* segment) but serve a different purpose. They are really just extra PC´s that can be set and maintained for the purpose of establishing offsets from some fixed address in an area outside the three segments. Generally they are used in conjunction with storing information on a stack. Symbols declared in these segments can be absolute or relocatable, depending on the attributes of the operand. Symbols declared in a *struct* segment can be reused just as if they had been defined using the *set* directive. Symbols declared in a *base* segment may be used only once, like any other label. A short example program may be the best illustration:

```
stack equu     $EF0000

      base     $000000
temp  ds.w     1
saved ds.l     2
junk  ds.b     1

      text
start move.l    stack,a0
8     move.l   junk(a0),d2
      add.l    temp(a0),d2
      move.l   d2,saved(a0)
      bne.s    8b
      end
```

In this example, the *base* directive allowed us to set up the variables *temp*, *saved*, and *junk*, which are offsets from the base location of the stack. Had a *struct* directive been used in place of the *base* directive, we could have reused the variables *junk*, *temp* and *saved* in other stack structures. The *struct* directive is extremely useful in defining stack structures in subroutines, where names such as *ret_address*, *frame_ptr*, *arg1*, etc., can be used over and over again without conflict. These directives do not actually create a segment, they merely set up a new PC which can be temporarily used to establish offset variables. These directives have most of the same permissions and restrictions as the *bss* segment; they default to location $000000 if first called without an operand. No code may be generated while the *base* or *struct* PC's are in effect, and new *base* or struct addresses are allowed. The segments end when a new segment begins.

global

The *global* directive is used in relocatable modules to inform the assembler that the symbols declared global should be passed on to the linking loader. The syntax of the *global* directive is:

global <label1>[,<label2>, . . .]

Label1, *label2*, etc. represent the symbolic names of the labels to be declared as global; each label should be separated by a comma. The global directives must occur before the use or definition of the symbol. Normally, global symbols are declared at the beginning of the source module. Local labels cannot be declared global.

Define and Enddef

These convenience directives work much the way *global* works. The *define* directive informs the assembler that all labels declared in the label field are declared as global symbols. This *define* mode is in effect until a *enddef* directive is encountered. For example,

```
data
define
templ    fdb      0,$FFFF
start    move.l   1,d1
enddef
```

This example simply defines the two labels *templ* and *start* as global. This directive works well when many symbols must be declared as global while they are initialized to various values.

Extern

The *extern* directive declares symbols to be external to this particular module. Local labels cannot be declared external. The syntax of the *extern* directive is:

```
    extern <label1>[,<label2>, . . .]
```

Label1, label2, etc. are ordinary labels as in *global;* labels should be separated by a comma. When the assembler encounters a label declared external in the operand field, external records are written out to the binary output module. With the *global* directive, the *extern* directive should appear before the actual use of the external symbol, usually at the beginning of the source module. These external records are used by the linking loader.

Name

Each binary output module can be given a module name with the *name* directive. The module name is used by the linking loader in reporting errors and address information; it is strongly recommended to give each module a name. The syntax of the *name* directive is:

```
    name <name of the module>
```

The module name can be a maximum of 14 characters. If more than one *name* directive occurs in the source module, the last name given will be the name assigned to the module.

Common and Endcom

It is possible to establish common blocks in the assembler. These can only be named and uninitialized common blocks.

```
    <name> common
```

A common block declaration is terminated by the use of the *endcom* directive. The only directives allowed between the *common* and *endcom* are *rmb* and *ds,* which define the size of the common block. Labels may be associated with each *rmb* or *ds* within the common block. For example:

```
test     common
temp1    ds.w   5
temp2    ds.l   10
         endcom
```

This common block is named *test,* has two variables (*temp1* and *temp2*) associated with it, and is 50 bytes long.

A common block and its variables are considered external by the assembler. Only one common block of a particular name should appear in a module. Because common blocks are treated as externals, the linking-loader handles the resolution of references to the common blocks automatically. For example:

```
         text
test     common
temp1    rmb  4
temp2    rmb  2
         endcom
start    move.l temp1,d0
         . . .
```

Common blocks are useful for passing parameters and keeping common information around. Furthermore, the common block will have the name of the common block as its module name; this is done automatically by the assembler. Common blocks must be accompanied by executable code in the same module that is, a common block cannot be the only item in a single source module.

ERROR AND WARNING MESSAGES

The assembler issues two types of error messages: fatal and non-fatal. A fatal error is one such as a disk file read error, which causes an immediate termination of the assembly. A non-fatal error results in an error message being inserted into the listing and some sort of default code being assembled if the error is in a code producing line. The assembly is allowed to continue on non-fatal errors. Error messages may not be suppressed.

All messages are output as English statements, not as error numbers. These messages announce violations of any of the rules and restrictions set forth in this manual and are essentially self-explanatory. Non-fatal error messages are preceded by with three asterisks, making them easy to locate.

Fatal error messages are sent to the standard error output. They are issued in the form:

Last Line = <last_line_read>
Line Number <line_num>
Fatal Error - <error_message_reported_here>

The messages that may come in the third line are listed later in this section. The last line processed is not reported on read, write, open or seek errors.

Possible Non-Fatal Error Messages

```
        16-bit expression expected.
```

A 16-bit expression was required in the operand field and the expression found cannot be represented in 16 bits.

```
        8-bit expression expected.
```

An 8-bit expression was required in the operand field and the expression found cannot be represented in 8 bits.

```
        A label declared "global" was not found in the program.
```

All labels declared via the *global* directive must be defined in the module.

```
        Absolute expression required.
```

An absolute expression is required in this context.

```
        Branches not allowed across segment boundaries.
```

Branches cannot be made to labels in other segments or to externals.

```
Can't subtract two relocatables from different segments.
```

Subtraction of relocatables is not allowed if they are from different segments.

```
Couldn't evaluate expression.
```

The expression could not be evaluated.

```
Couldn't evaluate expression in pass1.
```

Assembler directives such as *ds* and *rmb* must be evaluated in both passes of the assembler. Only a constant operand is legal, and forward references are not allowed.

```
Couldn't find that local label.
```

The local label specified in the expression was not defined. Note that the local labels "0" and "00" are two distinct labels.

```
Data register required.
```

A data register is required as one of the operands for the instruction specified.

```
Duplicate label found.
```

The label on this line has been defined more than one time.

```
Evaluator : attempt to divide by zero.
```

The divisor of an expression evaluated to zero.

```
Evaluator : more than one external found in an expression.
```

Only one external variable can be used per expression.

```
Evaluator : must shift by positive, non-zero quantity.
```

Only non-negative shift amounts are legal.

```
Evaluator : not a valid operation for 2 reloc's or extern's.
```

The assembler detected an attempt to add to relocatables or externals. Only absolute expressions can be added to externals or relocatables.

```
Evaluator : operator only valid for absolutes.
```

The following operations can be performed on absolute expressions only: and, or, exclusive or, not, multiply, divide, shifts and the logical operators.

```
Evaluator : unbalanced expression (wrt segments).
```

The expression evaluator found an expression involving relocatables from different segments. In expressions containing relocatables, the relocatables must be paired and canceling. A relocatable expression can only be relocated relative to one segment.

```
Evaluator : unbalanced parenthesis.
```

The parentheses in the expression were not balanced properly.

```
External expression not allowed.
```

An external expression is not allowed in this context.

```
External symbol not allowed in this context
```

In some of the directives, an external symbol is not allowed. For example, the *equ* cannot have an external symbol in the operand field; a symbol cannot be equated to an external symbol.

```
Extra arguments found.
```

Only two operands were expected for this opcode, but more were found.

```
Found zero branch length on short branch.
```

A short branch cannot be made to the immediately following instruction.

```
Forced short but long expression found.
```

The expression which was forced short (via *:W*) could not be fitted into a word.

```
IFDEF contained expression that couldn't be evaluated in Pass 1.
```

In conditional assembly, the assembler must be able to evaluate the condition in both passes. This expression can therefore not involve a forward reference to any variables.

```
Illegal addressing mode for instruction.
```

An addressing mode (specified in an operand) was is not legal for this instruction.

```
Illegal character in label.
```

Labels must consist only of alphabetic characters, digits and the underscore character.

```
Illegal expression or missing operand.
```

An expression could not be successfully parsed by the expression evaluator.

```
Illegal nesting of conditionals has occurred.
```

Conditional assembly rules have been broken. Conditionals can only be nested 20 levels deep and certain rules apply to their use. See the discussion of conditional assembly for a detailed description.

```
Illegal op-code for this segment.
```

Certain instructions cannot appear in some segments. For instance, no code can be generated in the BSS segment.

```
Illegal operand.
```

An error has been detected in the operand field.

```
Illegal register list for "movem".
```

The register list specified could not be interpreted. See the discussion of the instruction set for details on register list specification.

```
Illegal size for instruction.
```

The size specified by the .b/.w/.l extension is not allowed for this instruction.

```
Illegal special register for instruction.
```

The special register (USP,CCR,SR,VBR,...) specified as an operand is not legal for this instruction.

```
        Immediate size does not match instruction size.
```

The immediate operand was larger than the size specified in the instruction, or implicit in the instruction.

```
        Instruction expects only one operand.
```

The instruction specified has only one operand but more than one was found.

```
        Invalid binary header flag.
```

The operand of the *bhdr* directive is not a known binary header flag. The legal binary header flags are:

Executable	$04
Relocatable	$05

```
        Invalid local label - 0 thru 99 only.
```

Local labels must be in the range 0 through 99. Local labels may be reused in the same module.

```
        Invalid option specified.
```

The only legal options to the *opt* directive are *con, noc, lis, nol.* See the discussion of pseudo-ops and directives for more details.

```
        Invalid transfer address found (external).
```

External transfer addresses are not supported.

```
        Label required.
```

The directives, set, equ and log require a label to be specified on the same line.

```
        Negative value not allowed.
```

A negative value cannot be specified in this instruction.

```
        Nested COMMON's not supported.
```

Common blocks cannot be nested.

```
No closing delimiter found.
```

The assembler found the EOL character before finding a closing delimiter in a string expression.

```
No ENDCOM directive found.
```

A common block declaration must be bracketed by the two directives *common* and *endcom*. Another segment was entered without an *endcom* being specified.

```
Odd branch address found.
```

A branch to a label on an odd address was detected. Instructions must always begin on an even boundary. (The assembler takes care of this.) This can happen if a label is on a line by itself after some odd number of bytes of data, or the label is on a line with data that does not need to be aligned.

```
Overlapping register list specified.
```

The register list in the *MOVEM* instruction contains registers that have been specified more than once. The assembler issues this more as a warning than as an error, but the register list should be corrected.

```
Phasing Error.
```

The two passes of the assembler do not agree on the address of the label on the current line. This error can be caused by other errors in the assembly and should not appear as the only error in a given module. Only the first phasing error is reported, and checks are made only on lines containing labels.

```
Quicknumber (1-8) expected.
```

The instruction specified requires a *quick* count in the immediate operand field, and the expression found was not between 1 and 8 inclusive.

```
Relocatable displacement from the same segment required.
```

For PC relative code, the relocatable displacement must be from the same segment as the PC.

```
Relocatable displacement not allowed.
```

A relocatable displacement is not allowed in this context.

```
Relocatable expression required.
```

A relocatable expression is required in this context.

```
Symbol found in 'extern' also found as program label.
```

A symbol declared external to a module via the *extern* directive cannot be defined in the same module.

```
The string was too long for the size specified.
```

The size specified in the instruction is smaller than the size of the immediate string specified as the first operand.

```
Too far for a branch instruction.
```

The target of a branch instruction must be within the constraints of a 16-bit expression. A jump will have to be used.

```
Too far for a short branch.
```

The target of a short branch must be within the constraints of an eight-bit expression. A long branch will have to be used here.

```
Undefined symbol found.
```

A symbol in an expression has not been defined.

```
Unknown addressing mode.
```

The addressing mode specified could not be interpreted by the assembler.

```
Unknown opcode.
```

The opcode on this line is not a known opcode. See the discussion of instructions and format for details.

```
Unknown size specified.
```

The only legal size extensions on instructions are "s", "b", "w", and "l." A size other than this was specified.

```
Word operand required on system call name.
```

The system call specified is not a legal system call. The system call number must fit in 16 bits (word). See Section 6, *System Calls* for more information.

Possible Fatal Error Messages

```
Library file <file_name> not found
```

The specified library file could not be located. The assembler searches first in the current directory, then in a directory called *lib* in the current directory, and finally in the directory */lib*.

```
Library nesting too deep
```

Libraries may be nested only up to seven levels deep.

```
Local label table overflow
```

The maximum number of local labels allowed in a source file is 500.

```
No file specified
```

The assembler found no source files on the command line.

```
Opening <file_name>: <reason>
```

The assembler received an error from 4400 while opening the specified file. An explanation of the error message is given.

```
Out of space
```

The assembler's symbol table is grown dynamically and grew to the limits of the size restrictions imposed by the 4400. The solution is to break the source into multiple modules and assemble them separately.

```
Reading <file_name>: <reason>
```

The assembler received an error from the operating system while reading the specified file. An explanation of the error message is given.

```
Seeking in <file_name>: <reason>
```

The assembler received an error from the operating system while seeking in the specified file. An explanation of the error message is given.

```
U requires label
```

The "U" option requires a label as its argument. See the section on options for more details.

```
Unknown option '<char>'
```

The character specified is not a known option.

```
Writing to <file_name>: <reason>
```

The assembler received an error from the operating system while writing to the specified file. An explanation of the error message is given.

THE LINKING LOADER

Terminology

The remainder of this section describes the linking loader. The following additional terms are used:

loading	The placement of instructions and data into memory in preparation for execution. This preparation includes linking (the matching of symbolic references and definitions), and relocation of symbols and address expressions.
module	A subprogram which has been assembled using the assembler.
module name	The name given to a module by the programmer by using the *name* directive of the relocating assembler. If the *name* directive was not used, the module name is the same as the file name in which it is contained. Therefore, several modules may have the same name. The output module of the loader may be given a name by use of the "N" option.
relocatable object-code module	Equivalent to *module.*

Linking Loader Input

The Linking Loader accepts independently assembled, relocatable object-code modules as input. Relocatable object-code is generated by the assembler *asm* in such a way that addresses are not bound to absolute locations at assembly time; this binding of the address fields is accomplished by the Linking Loader. The *load* command binds the addresses at the time the object-code segments are combined to produce an executable program. The binding or adjustment of the address fields is termed *relocation.* Relocation is necessary when an instruction expects an absolute address as an operand. The address field of this instruction must be increased by a *relocation constant.* The relocation constant is the address where the module is loaded for execution.

Address fields which do not require relocation are absolute addresses; their values remain the same regardless of the position of the object code segment in memory. Since the loader does not have access to the source text, it cannot determine if an address field is absolute or relocatable. In fact, it cannot distinguish addresses from data or opcodes. Therefore, the assembler must indicate to the loader which address fields require relocation. This communication is accomplished through *relocation records,* which are appended to the object-code file produced by the assembler. Such a file is called a *relocatable object-code module.*

Often it is desirable for parts of a program (called modules) to be developed separately. Each module must be assembled separately prior to final merging of all the modules. During this merging process, it is necessary to resolve references which refer to addresses or data defined in another module. The resolution of these *external references* is called linking. The assembler must provide information to the loader, in a manner similar to relocation records, concerning the address fields which must be resolved.

Linking Loader Output

As output, *load* produces an object-code module, a load map, a module map, and a global symbol table. The object-code module can be either relocatable or executable. A relocatable module produced by the loader cannot be distinguished from a relocatable module produced by the assembler. Only the loader, however, can transform multiple relocatable modules into an executable program.

The Standard Environment File

An environment file, */lib/std_env* is supplied with every 4400. The loader uses this file to get the information necessary to load a given module. The environment file is just an options file (described earlier in this section) which is processed before any other options on the command line. This file contains hardware-specific information so the user will not need to specify these things each time a file is loaded. Information such as the hardware page size and the starting address of the text or data segments is typically found here. The */lib/std_env* options file is also linked to */lib/ldr_environ*.

Invoking the Loader

The linking loader accepts as input previously assembled, relocatable object-code modules and produces as output either:

1. A link-edited, relocatable, object-code module or
2. A link-edited, relocated, executable program.

The command line necessary to invoke the linking loader is:

```
        ++ load <relocatable_modules> [+options]
```

The two plus signs are the system's ready prompt, and *load* is the name of the linking loader command file.

<relocatable_modules> is a list of one or more file names, separated by blanks, which contain relocatable object-code modules you wish to load. The object-code modules are loaded in the order specified.

Options is a list of options which must start with a plus sign ("+") and may not contain any embedded spaces. More than one list of options may be specified, but each list must start with a plus sign. Some of the options are single characters, while others require an argument. Those that are single letters may be grouped together; for example: +sm. Those that require arguments may either stand alone or be the last of a group of options; for example: +smT=400000 (the *T=400000* is an option with an argument). The equal sign is not required in options with an argument. Therefore, *+T=400000* is equivalent to *+T400000.*

Valid Options

+a=<minimum_number_of_pages> The "a" option specifies the minimum number of pages to be allocated to this task when executing. The number of pages specified must be a non-negative decimal number up to the maximum of 32767.

+A=<maximum_number_of_pages>

The "A" option specifies the maximum number of pages to be allocated to this task when executing. The number of pages specified must be a non-negative decimal number, and should be greater than or equal to the minimum specified. The loader automatically adjusts *ridiculous* page counts. The maximum number of pages that can be specified is 32767.

+b=<maximum_logical_task_size>

The "b" option tells the operating system the largest size that this task may grow to while executing. The maximum logical task sizes currently supported are:

128K
256K
512K
1M
2M
4M
8M

For example :

```
+b=512K
+b=2M
+b=8m
```

The letters "M" and "K" may be in upper- or lowercase. The default task size is 128K. The loader automatically adjusts the task size if it finds that the size specified by the user or the default size is too small for the modules being loaded.

+B Do not zero the BSS space.

+c=<source_module_type>

The "c" option allows the user to specify from what type of source file this module was created. This information is placed in the binary header for use in debugging. The source module types currently recognized are:

ASSEMBLER
C

and are specified as follows:

+c=ASSEMBLER
+c=C

Upper or lower case letters may be used.

+d This option specifies no core dump is to be produced.

+D[=<start_of_data_segment>]

The "D" option specifies the *data* segment bias to add to all *data* references (i.e., the starting address of the data segment). The number specified as the start of the data segment must be in hex and is machine dependent. If no starting data address is given, the data segment follows the text segment. The "D" option with no argument forces the data segment to follow the text segment (the default). This may be necessary if the *std_env* file contains data and/or text starting addresses and the user wishes to load a module for execution on another machine with different hardware requirements.

+e By default, the loader notifies the user only once about each unresolved external symbol. This option forces it to report every occurrence of every unresolved external symbol, showing in which module it was unresolved.

+f This option loads the text page when it is first referenced. (Load on demand.)

+F[=<options_file_name>]

This option allows the user to place loader command line options in a file rather than listing them each time on the command line. The file is read by the loader, and options are set from there as well as from the command line. The last occurrence of an option always overrides previous occurrences, so if the options file is specified first, any options found on the command line will override the same option in the options file. Nested options files are not supported.

The options specified in the options file must be separated by one or more spaces, may be on multiple lines, and must begin with a "+" just as they do on the command line. Only options may be specified in this file, not modules to be linked. The loader discards all characters up to a "+," so comments may be inserted before the first option on a given line. For example, the following is a valid options file.

```
* Tektronix 4400 environment *
* Machine configuration +C=4
* Text Segment offsets +T=0
* Data Segment offsets +D=0
* Page Size default (hexadecimal) +P=1000
* System call +U=TRAP15
```

If the "F" option is specified without an argument, the loader looks for the file called *ldr_opts* in the current directory, and uses it as the option file.

+i

The "i" option includes all internal symbols in the symbol table for symbolic debugging. If the "i" option is not specified, only global symbols are included in the relocatable, object-code module.

+l=<library file name>

A maximum of five libraries may be specified by repeated use of the "l" option. If fewer than five libraries are specified, the system library is also searched in addition to the user libraries. Libraries are searched only when an executable output program is produced (not when "r" is specified). In the following example, an effort is made to resolve externals not found in the user's modules by searching the three libraries *lib1, lib2,* and *Syslib:*

```
++ load echo[1-3].r +l=lib1 +l=lib2
```

For more information concerning the formation and use of libraries, see the discussion of libraries later in this section.

+L

Do not search the libraries for unresolved externals.

+m

Print the load map and the module map. The load map provides information as to the type of output file produced, the length of the resulting output object-code module, the number of input modules, and the transfer address. The module map describes the load address and object-code length for each input module.

+M=<map output file>

The load map, module map and symbol table are written by default to standard output. This option specifies a file name to which this information is to be written, rather than standard output.

+n

Produces an executable output module with totally separate instruction and data spaces. This option informs the operating system that hardware support for separate instruction and data spaces is available and to handle addressing accordingly.

+N=<module name>

Specifies the name to be given to the output module of the loader (in a manner similar to the *name* directive of the assembler). Since the loader does not propagate the module names of the relocatable input modules to the output module, the "N" option must be used to assign a name to a module. If the "N" option is not used, the module name will default to the name of the file in which it is contained. Both executable programs and relocatable modules can receive module names. The name is limited to a maximum of 14 characters.

+o=<file_name>

Specifies the file name for of the output binary file. If the "o" option is not specified, the output file will be named *file_name.o* in the current directory. If a file by this name already exists, it will be deleted.

+P=n

This option specifies the page size to use. "n" is given in hex, with a default of $0.

+r

The "r" option specifies that the loader's binary output is to be a relocatable object-code module. The effect of this option is as if all the modules were contained in one source file and assembled with the assembler.

+s

The "s" option directs the loader to print the global symbol table. If specified, the loader prints each global symbol and its address.

+S=<initial_stack_size>

This option informs the operating system that a task needs some amount of stack space when it begins execution. The operating system, by default, sets up a 4K-byte stack for each user task. This mechanism allows the task to begin execution with possibly more than the normal amount of stack space (but never less — the operating system will always round up to the next 4K-byte boundary).

+t

The "t" option specifies that the loader's binary output is to be a shared text, executable program. For more information, see *Shared Text Programs* in the discussion on *segmentation*.

+T=<start_of_text_segment>	
	This option specifies the *text* segment bias to be applied to all text references. The starting address must be in hex; it defaults to 0 when the "T" option is not specified. The text bias is machine dependent.
+u	This option tells the loader not to print the *unresolved external* message when producing a relocatable output module.
+U=n	This option sets the trap number for system calls.
+x=<file_name>	This option declares the incremental load file name.

Libraries

Introduction

A library is a special collection of relocatable modules. When an external cannot be resolved from the user's modules, the libraries are searched in an effort to resolve it. The loader searches the user defined libraries in the order specified on the command line before searching the system library. This allows the user to redefine a system library module or entry point. The search for an external can be summarized as follows:

1. Can the external be resolved from the user's modules?
2. Can it be found in the user specified libraries?
3. Can the external be resolved from the system library?

When an external is resolved from a module contained in a library, that module is loaded and is then considered to be a *user* module. Because of this, library modules can reference other library modules.

The loader can search a maximum of five libraries when externals cannot be resolved from the user's modules. Usually, these libraries consist of up to four user libraries and the system library. The user can, however, specify five libraries on the command line. When five libraries are specified, the fifth one takes the place of the system library.

When searching for a library, the loader first looks for the specified file in the current directory. If the file is not found, the loader then looks for the *lib* directory in the current directory. If it finds that directory, the loader attempts to find the specified file. If not found, the loader makes a third and final attempt to find the specified file by looking in the directory */lib.* If the file is not found in any of these three directories, an error message is issued and the loader aborts. This process also is followed when searching for the system library.

Library Generation

The *libgen* utility is used to create new libraries and update existing libraries. All modules in a library **must** have a name. The name is assigned to a module by the *name* pseudo-op in the assembler or by the "N" option of the loader. It is the responsibility of the programmer to ensure that all modules in a library have names. The *libgen* utility will not accept a module without a name.

The syntax for the *libgen* utility is:

```
        libgen o=<old> n=<new> u=<updates> <options> <deletions>
```

The arguments may be specified in any order.

The argument *o=<old>* specifies the name of an existing library file. This library file must have been created previously by *libgen*. If *libgen* is being called to create a new library (instead of updating an existing one), this argument should be omitted.

The argument *n=<new>* specifies the name of the new library. If this file already exists, it is deleted before the new library is written. This argument need not be specified when updating an existing library. In this case, *libgen* will put the new library in a scratch file, delete the old library file, and rename the scratch file, giving it the name of the old library. The command line must include either the *o=<old>* or *n=<new>* argument, or both.

The argument *u=<updates>* specifies the name of a file containing modules that are to be added to the library, replacing existing modules in the library if necessary. More than one update file may be specified by repeating the *u=* argument. Up to nine files may be specified in this way.

As *libgen* runs, it produces a report, describing the action that it has taken for each module in the library. The report includes the module name and the file from which it was read (the old library or one of the update files). The options are used to eliminate or shorten this report. If the "+l" option is specified, no report is produced. If the "+a" option is specified, the report only contains information about those modules that were replaced, added, or deleted. No information about modules copied from the old library is given.

The *<deletions>* argument is a list of module names to be deleted from the old library. The names may be separated by commas or spaces. If a name is specified that cannot be found in the old library, a warning message is issued. If the "+l" option was specified, no warning is issued.

Examples

1. Create a new library with the name *binlib* containing modules from the files *one*, *two*, and *three:*

   ```
   libgen n=binlib u=one u=two u=three
   ```

   Since a new library is being created, the *o=<old>* argument was omitted. Note that the *u=* argument was repeated for each update file.

2. Update the library named *binlib*, adding or replacing records from the file *new*. Produce an abbreviated report:

   ```
   libgen o=binlib u=new +a
   ```

   Since no new library was specified, the new library is given the name of the old library.

3. Update the library named *binlib*, deleting the modules named *diagonal* and *transpose;* add new modules from the file *xyz* and write the new library in the file *newlib:*

   ```
   libgen obinlib u=xyz n=newlib transpose diagonal
   ```

Segmentation and Memory Assignment

Relocatable and Executable Files

The loader can produce either an executable program or a relocatable module. By default, the loader produces an executable module: use of the "r" option causes the loader to produce a relocatable module. The loader can produce two types of executable files: shared text and non-shared text. The next sections will discuss how *load* produces the relocatable modules and the two types of executable programs.

Relocatable Modules

Relocatable modules produced by the assembler have distinct text, data, and bss segments. All of the text object-code appears in the binary file first, followed by all of the data object-code. Since there is no object-code in the bss segment, it is thought of as following the data segment. The loader maintains these distinct segments by combining the text segments of all the relocatable input modules, followed by the concatenation of data segments, and then (conceptually) all the bss segments. In addition, the module segments are loaded in the order in which the modules are specified on the command line.

Common blocks (which contain only bss) are not combined with the bss segments of the other modules when producing a relocatable output module. Instead, common blocks retain their identity as separate modules and are appended to the resulting relocatable output module. Common areas are combined with the bss segments of other modules only when producing an executable program.

Relocatable modules can be given module names by the use of the *name* directive of the assembler. This name is used when printing the module map. If no name was given to a module by use of the *name* directive, the name of the file in which it is contained is printed. When producing a relocatable output module, the loader does not propagate any of these module names to the output. To assign an output module a name, use the "N" option when invoking the loader.

Unlike module names, *info* fields are collected from the input modules and carried over to the relocatable output module and ultimately to the executable program.

Executable Programs

When loading modules to produce an executable program, the loader loads modules in the order specified on the command line. Common areas (that contains only uninitialized data) are loaded after the last module specified on the command line. Libraries are loaded after the last common block, or after the last user module on the command line if there are no common blocks.

The two types of executable programs the loader is capable of producing are shared text and non-shared text.

Shared Text Programs

Shared text programs have three distinct segments. The *text* segment is assumed to be read-only. This implies the code contained in the text segment is not altered as long as the program runs. We can take full advantage of this fact by *sharing* this segment among several users who are running the program concurrently. This can mean a considerable increase in the efficiency of the system.

The *data* segment is also referred to as *initialized data.* It is information (actual instructions or data) that must be initialized or loaded, but which can be altered at some later point. For example, counter variables which must be initialized to zero but will later be incremented should be placed in the data segment. At any time, the variable could be read or its value changed. Each user would then need his or her own copy of the data segment.

The *bss* segment, like the data segment, is also a read/write area. Since a module does not contain any object code to be loaded into this section of memory, it is also referred to as *uninitialized data.* The module does contain the size of the bss segment, however, in order to inform the operating system that memory is required in this area but does not need to be initialized.

When producing a shared text program, the loader collects all the text segments from the relocatable input modules and loads them at the location specified by the "T" option, or at 0 if no starting text address was given. All of the data segments are then placed either at the address specified in the "D" option or, if no "D" was given, immediately following the last text address, rounded up to the granularity specified in the "P" option (or the next even byte if no "P" was specified). Memory for the bss segments is allocated immediately following the data segments at the time the program is executed.

There are drawbacks to using shared text. The text portion of a shared text file is always swapped to disk. Therefore, programs which are used infrequently, or those that only one task would be running at a time, would make better use of the system resources if they were non-shared.

The following memory map illustrates how the segments are loaded in relation to other segments and modules. The module numbers are the order in which they appear on the command line; "m" is the last module specified. Common blocks 1-x and library modules 1-n, which are loaded to complete the program, are also represented.

```
Hardware
Dependent --> Text of mod 1
              Text of mod 2
                    .
                    .
              Text of mod m
              Text of library 1
              Text of library 2
                    .
                    .
              Text of library n
                          <--+
                             !  Depends on 'P' option.
                             !  Hardware Dependent.
                          <--+
Hardware
Dependent --> Data of mod 1
              Data of mod 2
                    .
                    .
              Data of mod m
              Data of library 1
              Data of library 2
                    .
                    .
              Data of library n
              Bss of  mod 1
              Bss of  mod 2
                    .
                    .
              Bss of  mod m
              Bss of common 1
              Bss of common 2
                    .
                    .
              Bss of common x
              Bss of library 1
              Bss of library 2
                    .
                    .
              Bss of library n
```

Non-shared Text Programs

A non-shared text program has the same form as shared text program except that it is simply not shared. The non-shared text programs do not incur the overhead of having their text segments swapped immediately to disk at execution time. The memory map for a non-shared text program is the same as for a shared text program.

Load and Module Maps

Load Map

The "m" option controls the printing of the module and load maps. The load map provides information about the type of output produced, the length of the resulting output object code module, the number of input modules, and the transfer address.

Module Map

Use of the "m" option also selects printing of the module map. The module map describes the load addresses and object code length for each of the input modules.

The Module Map of a Relocatable Module

When producing a relocatable module, both the assembler and the loader do not *bind* or tie addresses to absolute locations; they are made relative to the base of the segment to which they refer. The following example assembled by the assembler illustrates this point.

```
1                                  extern   pdata
2  000000                          text
3 +000000 207C 0000 0000 Start     move.l   #msg1,a0
4 X000006 4EB9 0000 0000           jsr      pdata
5 +00000C 207C 0000 000A           move.l   #msg2,a0
6 +000012 23C8 0000 0012 lab1      move.l   a0,msgaddr
7 X000018 4EB9 0000 0000           jsr      pdata
8  00001E 4E75                     rts
                    * Start of DATA segment
10  000000                          data
11  000000 4D65 7373 6167 msg1      fcc      "Message 1",0
12  00000A 4D65 7373 6167 msg2      fcc      "Message 2",0
                     * Start of BSS segment
14  000000                          bss
15  000000                          rmb      18
16  000012                 msgaddr  rmb      4
                     * Set transfer address
18          0000 0000               end      Start
```

All of the segments start at address 0 (lines 2, 10, and 14). This is called the base address. Because of this, it is possible for two labels in different segments to have the same address (offset from the segment base). *lab1* and *msgaddr* are examples of this occurrence. All labels defined in a segment are relative to its base address. For example, *lab1* is 18 bytes from the beginning of the text segment, and *msg2* has an offset of 10 bytes from the base of the data segment. Throughout the linking process, the distance between *start* and *lab1* will remain constant. No assumptions, however, can be made about the distance between two labels that reside in different segments.

To produce a relocatable module from several input modules, the loader must combine all like segments. In other words, all text segments are concatenated starting with the text segment of the first input module, followed by the text of the second module, and so on. By doing so, however, the base address of all modules except the first will be changed. The loader automatically adjusts any addresses which refer to symbols in these modules which have been *relocated.*

A small "C" program was compiled, assembled and loaded, producing the following load and module maps:

```
* LOAD MAP *


Produced - executable, not overlapped TEXT and DATA.
Module is not shared text.
Starting TEXT address = 000000
Starting DATA address = 400000
Initial stack size = 000000
Granularity = 000000
Binary transfer address = 000B38
Number of input modules = 5


* MODULE MAP *


TEXT    DATA    BSS       MODULE NAME       FILE NAME
000000  400000  400204    test              test.r
000050  40000C  400204    Long Mul/Div      /lib/Clib
00032A  40000C  400204    C System Calls    /lib/Clib
000B04  4000A0  400204    strlen            /lib/Clib
000B38  4000A0  400204    C Wrapper         /lib/Clib


000B52  400204  400604    * Final Segment Addresses *
```

The maps show the text segments from each of the modules are combined and relocated to form the text segment of the final executable module. The starting text address was specified as 0. The starting data address was 400000. From looking at the module map we can see that all modules have text segments, the *Long Mul/Div* and the *strlen* modules have no data segments, and only the *C Wrapper* module has bss. Note also that the bss segment follows immediately after the data segment. The library */lib/Clib* was searched successfully for the routines called directly and indirectly by the module *test.* One can see that the binary transfer address is located in the *C Wrapper* module (Address $000B38).

The following map was produced using the same file as the previous map. No starting address for the data segment was specified, therefore the data segment follows the text segment. Since a granularity was specified as $1000 (The "P" option), the data segment starts at the end of the text segment (rounded up to the next $1000 boundary). The executable module is to be shared text.

```
* LOAD MAP *


Produced - executable, not overlapped TEXT and DATA.
Module is shared text.
Starting TEXT address = 000000
Starting DATA address = 000000
Initial stack size = 000000
Granularity = 001000
Binary transfer address = 000B38
Number of input modules = 5


* MODULE MAP *


TEXT    DATA    BSS       MODULE NAME      FILE NAME
000000  001000  001204    test             test.r
000050  00100C  001204    Long Mul/Div R   /lib/Clib
00032A  00100C  001204    C System Calls   /lib/Clib
000B04  0010A0  001204    strlen           /lib/Clib
000B38  0010A0  001204    C Wrapper        /lib/Clib


000B52  001204  001604    * Final Segment Addresses *
```

Miscellaneous

Transfer Address

A transfer address is the location at which execution is to start when the program is invoked. The *end* directive in the relocating assembler can be used to indicate a transfer address.

Only one relocatable module included in a program should contain a transfer address. If more than one module has a transfer address, the loader prints an error message and aborts.

Resolution of Externals With Library Modules

The loader resolves externals in the following manner:

1. Combine all user modules.
2. Search libraries sequentially resolving all references that the user modules make to the library modules. (Primary references)
3. Search libraries again, this time resolving any external references made by the library modules brought in during step 2. (Secondary references)

When resolving externals with library modules, the loader always processes the libraries in the order specified on the command line. When resolving secondary references (step 3 above), if bringing in another library module introduces more unresolved externals, then the library search begins from the beginning again. This way, even though the same module may appear multiple times in different libraries, only the first occurrence of each module (as defined by the order of the libraries on the command line) is used.

Etext, Edata, and End

In certain applications, it is desirable to know the last location contained in a particular program segment (text, data, or bss). Due to the manner in which these modules are loaded, it would be very difficult to determine these locations in an applications program. To alleviate this difficulty, the loader has three global symbols which are always available and contain the location of the end of a segment. These three globals are ETEXT, EDATA, and END; they correspond to the ends of the text, data, and bss segments respectively.

ETEXT, EDATA, and END may be used like any other user-defined global symbols. Since they behave like user-defined globals, they always appear in the global symbol table listing. When used in a module, they should be defined as external. These special symbols are pre-defined, so users should not give these names to their own global symbols.

Error Messages

The loader produces both fatal and non-fatal error messages. Fatal error messages are of the form:

```
Fatal Error: <description_of_error>
Loader aborted!
```

Non-fatal errors are produced in different forms for different messages.

Non-Fatal Error Messages

```
Warning: "/lib/std_env" not found.
```

The */lib/std_env* file is supplied with every 4400. It is an options file which contains hardware-specific information so that the user does not need to enter them for each load. If you have not deleted or renamed the file purposely, you should contact your Tektronix service representative.

```
<symbol_name> unresolved in module <module_name>.
```

The specified symbol was referenced in the specified module, but the symbol could not be located in any of the user supplied modules or in the libraries (if libraries are being searched). This may be expected if a relocatable file is being produced. If an executable file is being produced it is an error.

```
Symbol name clash: <symbol_name> in module <module_name>.
```

The specified symbol has been globally declared in more than one module. The module specified is the one containing the second declaration of the symbol. The name of the global symbol will have to be changed in one of the modules, and the module will have to be reassembled.

```
Integer overflow in module <module_name>.
Segment = <segment>.
Offset in module = <offset>.
```

When relocating a field in the module specified, the loader detected overflow out of the size field being adjusted. This may not always be an error. The address of the field relative to the specified segment is also reported. Subtracting from an external in a module can result in this message being produced when in fact the result of the subtraction is exactly as it should be. The user should look carefully at the code being loaded to determine if the error message should be ignored or not.

```
Two-Byte address overflow in module <module_name>.
Segment = <segment>.
Offset in module = <offset>.
```

This error message is similar to the preceeding one, but with one slight difference. A two-byte address (absolute word addressing mode from the assembler) must be a positive, 16-bit expression to be a valid address, whereas the previous overflow message requires only that the result be an unsigned 16-bit expression. This message definitely indicates an error. An address was forced to absolute short in the assembler when it can not be.

Fatal Error Messages

```
Illegal minimum page allocation!
```

The minimum page allocation must be a positive integer. The number specified on the command line is illegal.

```
Illegal maximum page allocation!
```

The maximum page allocation must be a positive integer. The number specified on the command line is illegal.

```
        Too many libraries!
```

A maximum of five libraries may be specified on the command line to the loader.

```
        Nested 'F' options!
```

Option files cannot be nested. Multiple option files can be specified on the command line though.

```
        Illegal configuration specified!
```

The configuration specified is not a known configuration. See the "C" option for more information.

```
        Illegal option <char>!
```

The character specified is not a known loader option. See the "options" discussion for more details.

```
        Relocatable, but data/text start specified.
        Conflicting options!
```

When producing a relocatable file as output, no starting text or data addresses can be given.

```
        Opening <file_name>: <reason>
```

The loader received an error from the operating system when it tried to open the specified file. An explanation of the error is given.

```
        Reading <file_name>: <reason>
```

The loader received an error from the operating system while trying to read the specified file. An explanation of the error is given.

```
        Writing to <file_name>: <reason>
```

The loader received an error from the operating system while trying to write to the specified file. An explanation of the error is given.

```
        Seeking to <location> in <file_name>: <reason>
```

The loader received an error from the operating system when it tried to seek to the specified location in the specified file. An explanation of the error is given.

```
        Unknown module type!
```

The module type specified on the command line is not a legal type. The loader only recognizes *C*, and *ASSEMBLER*. See the options discussion for more details.

```
        Illegal task size!
```

The task size specified on the command line is illegal. Allowable task sizes are: 128K, 256K, 512K, 1M, 2M, 4M, or 8M. See the options discussion for more details.

```
        No files given!
```

The loader found no files on the command line.

```
        Illegal input file <file_name>!
```

The specified file is not a legal relocatable file produced by the assembler or the loader.

```
        Library <library_name> not found!
```

The library specified could not be located in the current directory, a directory called *lib* in the current directory, or in the */lib* directory.

```
        Bad library format for <library_name>!
```

The library specified did not have the correct format for a library created by the *libgen* utility.

```
        Multiple transfer addresses!
```

Only one module can contain a binary transfer address. The loader found two user-specified modules with transfer addresses.

```
<file_name> contains MC68020 or MC68881 specific instructions.
```

The relocatable modules have MC68020 or MC68881 code and will not load/link on a MC68010 based product.

```
Invalid module combination - no output produced.
```

The relocatable modules have MC68020 or MC68881 code and will not load/link on a MC68010 based product.

```
Illegal relocation!
```

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

```
BSS instruction segment!
```

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

```
BSS transfer address!
```

This message is an internal consistency check and should not be issued. If this message is ever reported, contact your Tektronix service representative.

Section 4
SYSTEM CALLS

INTRODUCTION

Sections 2 and 3 provided an introduction to the 4400 system calls and the use of *asm* and *load*. This section describes each of the system calls, including errors that may be returned after the system call. This section is meant to be used with the assembler. If you want to make system calls from a high-level language, see the documentation for that language.

OVERVIEW

Assembly language programs on the 4400 interface to the operating system through system calls perform functions such as file manipulation and task control. The calls are implemented with the TRAP #15 opcode followed by a one-word function code which defines the call to be performed. Up to four 32-bit values (longs) may follow the function code, depending on the particular call. The 4400 assembler supports the *sys* pseudo-op which sets up the appropriate machine code for a system call. Its syntax is:

```
        sys     function[,arg0,arg1,arg2,arg3]
```

where *function* is the system call number or name. This pseudo-op produces the TRAP code for the call – a single word for the function and 32-bit values for each argument.

The arguments to system calls fall into three categories: numbers, pointers, and buffer addresses. Numbers may be bit patterns (as in the *chprm* call) or mode codes such as in *open*. A 32-bit value is used, even if the number required fits in 16 bits or less. Pointer arguments are used for calls that require a name or ASCII string (such as file names for *open* and *create*). The pointer is simply the address of the location of the string in memory. The string should always be null terminated (a 00 byte). A buffer address is used for calls, such as *status*, that require a place in the caller's address space to place data generated by the call. A buffer address is simply a 32-bit address pointing to the start of the data buffer. Some calls also extract data from a caller-supplied buffer.

Some system calls require information to be passed in registers as well as through arguments. Most calls use the D0 register, but a few use A0 as well. All registers are preserved through a system call unless a value is returned in the register. An error generated in a call always returns the error number in the D0 register.

Condition codes are also preserved through a system call with the exception of the error bit. The error bit is the same as the carry, and the assembler supports the *bes* and *bec* mnemonics *–branch if error set* and *branch if error clear*. These mnemonics are synonymous with *bcs* and *bcc*, respectively. The error bit always returns cleared if no error resulted from the call; otherwise, it is set and the error response code is in D0. The usage of each system call is described in a similar manner. To illustrate, here is an example of the *read* system call:

```
        <file descriptor in D0>
        sys read,buffer,count
        <bytes read in D0>
```

The information in the angle brackets preceding the call shows the data the system expects to find in the registers. In this example, the D0 register should contain the file descriptor number of the file to be read. Next is the actual system call as it would appear in the assembler source listing. The system function is *read* and it has two arguments: *buffer* and *count.* Following the call is information regarding the data to be found in the changed registers. In this example, the D0 register contains a count which represents the actual number of bytes read from the specified file. Other registers are unchanged.

NOTE

If a system call returns data to a buffer, it may not return it into the text *segment of a program; it may return the data to the* data *or* stack *segments. For example, the* buffer *in* read, *and* tbuf *in* time *may not reside in the* text *segment.*

NOTE

The ind, *system call signals an address error if the indirect target area is in the text segment. Keep the target area for indirect system calls in the data or stack segments.*

System Errors

When the system returns from a system call with the error (carry) bit set, register D0 contains the number of the resulting error. The file */lib/syserrors* lists all of the system errors and their corresponding error number. Here is a list of all system error numbers and their respective meanings:

1 EIO I/O error.

This can result from a CRC error, hardware malfunction, or defective media problem while reading or writing a device.

2 EFAULT System fault.

System faults are detected by the hardware and vary from system to system.

3 EDTOF Data section overflow.

This error can result from a *break* system call if the data section of a program is growing and overflows into the stack section.

4 ENDR Not a directory.

The file name specified is not a directory but the system call requires it to be one.

5 EDFUL Device full.

The device currently being written has no more available space.

6 ETMFL Too many files.

Each task is permitted a maximum of 32 open files at any one time.

7 EBADF Bad file.

The file descriptor given does not refer to an open file, or the file mode is not correct for the operation (e.g., the file is open for read and a write is attempted).

8 ENOFL No file.

The file name specified could not be found.

9 EMSDR Missing directory.

One of the directory elements specified in a pathname did not exist.

10 EPRM No permission.

An attempt was made to perform an action (such as file access) for which permission was denied.

11 EFLX File exists.

The system call requires the file to be previously non-existent.

12 EBARG Bad argument.

A bad argument was presented to a system call. This usually implies a number which is out of range or a non-existent mode code.

13 ESEEK Seek error.

An attempt was made to seek beyond the beginning of a file or beyond the physically possible maximum size of a file.

14 EXDEV Crossed devices.

An attempt was made to link to a file on a different device than the existing file.

15 ENBLK Not a block special file.

The file name specified was not a block special file, and the system call referenced requires it to be a block device (e.g. mount).

16 EBSY Device busy.

The device specified in an *unmount* is currently being used.

17 ENMNT File not mounted.

The file specified to an unmount call was not previously mounted.

18 EBDEV Bad device specified.

The system call requires a device type file as an argument.

19 EARGC Too many arguments.

Too many arguments were presented to an *exec* system call and the argument space overflowed. There is an upper limit of approximately 3000 bytes for arguments.

20 EISDR File is a directory.

The file specified is a directory, and the system call requires it to be a regular type file.

21 ENOTB File is not binary.

An attempt was made to execute a file that was not an executable binary file.

22 EBBIG Binary file too big.

The binary file specified to *exec* exceeds the physical address space limits.

23 ESTOF Stack overflow.

The stack space overflowed into the task's data or text space.

24 ENCHLD No children living.

A *wait* system call was executed with no living *child* tasks to wait for.

25 ETMT Too many tasks active.

In attempting to fork a new task, the system exceeded its task limit. This error will also result if the system task table becomes full.

26 EBDCL Bad system call.

A system call function code was encountered that does not represent an existing system call.

27 EINTR Interrupted system call.

One of the program interrupts that the current task was catching occurred during the system call.

28 ENTSK No task found.

The task id referenced in the system call did not represent an active task in the system.

29 ENTTY Not a tty.

The system call (*ttyget* or *ttyset*) requires the specified file to represent a tty type device.

30 EPIPE Write to broken pipe.

The system attempted to write data to a pipe that did not have an active read channel open.

31 ELOCK Record lock error.

The specified record can not be locked by this task. Another task has the requested record locked.

32 ETXOF Text segment overflow.

The program's text segment has exceeded the original specified size.

33 EVFORK Illegal operation in *vforked* task.

See *vfork* for more details.

34 EDIRTY Mounted disk is dirty.

The disk you attempted to mount was not unmounted before system shutdown. Run diskrepair to clean up the disk.

System Definitions

Several files containing system definitions reside in the */lib* system directory. Use these files as *library* files in the assembler whenever the appropriate definitions are required. Here´s a general description of each file:

syscomm	TTY buffer for the communications device. Similar for *systty,* but defined for the RS-232C host communications port.
sysdef	System call definitions. All of the system call names are defined in this file.
sysdisplay	System display and event function code definitions. This file contains the information returned by the *getDisplayState* system call, the equated function codes for vector calls and the bit positions within the status long word displayState record. zx
syserrors	System errors. All standard system error names and their equated error numbers appear in this file.
sysfloat	System floating point interface. All floating point routines and their operation codes are in this file. Also in this file is a sample general calling sequence for floating point operations.
sysints	System program interrupts. All program interrupt names are equated to their respective numbers in this file.
sysstat	File status block. This file contains the block definition for the information returned by the *status* and *ofstat* system calls.
systim	Time buffer definitions. The *time* and *ttime* system calls return their information in a caller provided buffer. These buffers are defined in this file.
systty	TTY buffer for the console device. The *ttyget* and *ttyset* require a buffer for their data transferal. The contents of this buffer is defined here.

DETAILS OF SYSTEM CALLS

set_high_address_mask

USAGE

```
<mask in D0>
sys set_high_address_mask
```

DESCRIPTION

The value <mask> will be used by the system to load the hardware address mask register. This hardware register masks the upper address bits from the processor. Each task has its own mask value which defaults to 0xFFFFFFFF.

NOTE

Bits 0-23 are forced to 0x00FFFFFF by the system.

DIAGNOSTICS

No errors are reported.

alarm

USAGE

```
<seconds in D0>
sys alarm
<previous seconds in D0>
```

DESCRIPTION

Alarm will cause an alarm interrupt to be issued after the number of seconds specified. At alarm time, the program interrupt SIGALRM is sent to the task. Unless this interrupt is caught or ignored, it terminates the task. This system call returns immediately to the caller after execution.

DIAGNOSTICS

No errors are possible from this call.

break

USAGE

```
sys break,address
```

DESCRIPTION

Break changes the amount of memory associated with the task. The *address* specifies the highest address to be used by the task for data. If the address specified is already in the assigned data space, any memory beyond it is released back to the system.

DIAGNOSTICS

An error is issued if more memory is requested than is physically possible on the system.

chacc

USAGE

```
sys chacc,fname,perm
```

DESCRIPTION

Chacc checks the accessibility of file *fname*. The *perm* argument should be *1* for read check, *2* for write check, or *4* for execute check. Any combination of these may be used (e.g. 3 checks read/write). If *perm* is 0, *chacc* checks if the directories leading to the file may be searched and if the file actually exists.

DIAGNOSTICS

Returns an error if the file does not exist, the directory path cannot be searched, or if the permission is not granted.

chdir

USAGE

```
sys chdir,dirname
```

DESCRIPTION

Chdir changes the current user directory to that specified by *dirname*, which points to the actual name. The caller must have execute permission in the specified directory.

DIAGNOSTICS

Issues an error if the name specified is not a directory or cannot be searched.

chown

USAGE

```
sys chown,fname,ownerid
```

DESCRIPTION

Chown changes the owner of the file name pointed at by *fname*. Ownerid should have a maximum of 16-bit significance. Only the system manager may execute this call.

DIAGNOSTICS

Returns an error if the caller is not the system manager.

chprm

USAGE

sys chprm,fname,perm

DESCRIPTION

Chprm changes the access permission bits associated with the file name represented by *fname*. The new permission bits *perm* will replace the old. The allowable permissions are:

```
FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
FXSET => %01000000 ($40)  set id bit for execute
```

DIAGNOSTICS

Issues an error if the file does not exist, or the caller is not the file owner or system manager.

close

USAGE

```
<file descriptor in D0>
sys close
```

DESCRIPTION

Close closes the file represented by the specified file descriptor. Files are automatically closed when the task that opened them terminates, but it is wise to close them manually whenever possible.

DIAGNOSTICS

Returns an error if the file descriptor is not valid, or if the file has already been closed

control_pty

USAGE

<master device file descriptor in D0>
sys control_pty,function,cval
<state in D0>

DESCRIPTION

This is the function used to control the behavior of a pesudo-terminal channel. The structure of the pseudo-terminals showing the equates is found in the file */lib/syspty* and */lib/include/syspty.h*. All functions return the state of the channel as described for the function *PTY_INQUIRY*.

The function *PTY_INQUIRY* is used to return the state of the channel. For this function, *cval* is ignored. The value returned is a combination of bits which describe the state of the channel. The bits are:

PTY_PACKET_MODE	Bit #0. If this bit is set, reads on the master side return two bytes of status before any data written by the slave. If any slave data is available, the status bytes are zero. If no data is present, the status bytes are the same as those returned by *PTY_INQUIRY*.
PTY_REMOTE_MODE	Bit #1. If this bit is set, data written by the master will be sent as is to the slave side with no editing.
PTY_READ_WAIT	Bit #2. If this bit is set, a read on the master side is blocked until slave data is available.
PTY_WRITE_WAIT	Bit #3. If this bit is set, the master side hangs on a write request if the output buffer is full.
PTY_HANDSHAKE_MODE	Bit #4. If this bit is set, a write on the master side is not complete until the slave has consumed the data.
PTY_SLAVE_HOLD	Bit #7. If this bit is set, the slave is prohibited from writing any more data to the channel.
PTY_EOF	Bit #8. If this bit is set all slave accesses to the channel have been closed.
PTY_OUTPUT_QUEUED	Bit #9. If this bit is set the slave side has written data to the channel which has not yet been consumed by the master.
PTY_INPUT_QUEUED	Bit #10. If this bit is set the master has written data to the slave side which has not yet been consumed by the slave.

The function *PTY_SET_MODE* is used to change the control mode for the pseudo-terminal channel. The value *cval* contains the new mode and should be some combination of the bits described in the previous section. The new control mode is exactly what is in *cval* so to perform an incremental change, the current value must be obtained using *PTY_INQUIRY*.

The function *PTY_FLUSH_READ* causes any data written by the master side to the slave input queue to be purged.

The function *PTY_FLUSH_WRITE* causes any data written by the slave side that has not yet been consumed by the master side to be purged.

The function *PTY_STOP_OUTPUT* prevents the slave side from writing any more data to the master side. This condition is reflected in the status bit *PTY_SLAVE_HOLD*.

The function *PTY_START_OUTPUT* allows the slave side to continue writing data to the master side.

DIAGNOSTICS

Issues an error if a bad file descriptor node is used.

cpint

USAGE

```
sys cpint,interrupt,address
<old address in D0>
```

DESCRIPTION

Cpint tells the system what action it should take when *interrupt* occurs. If the specified address is 0, the default action occurs (usually task termination). If the address is 1, the interrupt is ignored. An even address (not zero) is taken to be a valid user program address where control should be passed upon interrupt interception.

After interception, the interrupt number is in the D0 register. The user´s code should exit the interrupt code via an RTR instruction. Following the return, the task continues at the point it was interrupted.

After processing an intercepted interrupt, the system resets it back to the default condition; therefore, to continue catching the interrupt, it is necessary to re-issue a new *cpint* call each time the interrupt occurs. An exception is the SIGDEAD interrupt, which is not reset to a default condition after occuring. It should be noted that the SIGKILL interrupt cannot be ignored or caught. All interrupts retain their status after a *fork,* but *xec* resets all caught interrupts back to their default state. The system calls for *read* and *write* when referencing a slow device (like a terminal), and the calls *stop* and *wait* may return prematurely if a caught interrupt occurs during the system´s handling of them. If this happens, it looks as if the system call returned an error (EINTR), and the call can be re-issued if desired.

In the following list of system interrupts, those marked with "*" cause a core dump if not caught or ignored.

SIGHUP	1	Hangup
SIGINT	2	Keyboard
SIGQUIT	3*	Quit
SIGEMT	4*	EMT $Axxx emulation
SIGKILL	5	Task kill
SIGPIPE	6	Broken pipe
SIGSWAP	7	Swap error
SIGTRACE	8	Trace
SIGTIME	9*	Time limit
SIGALRM	10	Alarm
SIGTERM	11	Task terminate
SIGTRAPV	12*	TRAPV instruction
SIGCHK	13*	CHK instruction
SIGEMT2	14*	EMT $Fxxx emulation
SIGTRAP1	15*	TRAP #1 instruction
SIGTRAP2	16*	TRAP #2 instruction
SIGTRAP3	17*	TRAP #3 instruction
SIGTRAP4	18*	TRAP #4 instruction
SIGTRAP5	19*	TRAP #5 instruction
SIGTRAP6	20*	TRAP #6-14 instruction
SIGPAR	21*	Parity error
SIGILL	22*	Illegal instruction
SIGDIV	23*	DIVIDE by 0
SIGPRIV	24*	Privileged instruction
SIGADDR	25*	Address error
SIGDEAD	26	Dead child
SIGWRIT	27*	Write to READ-ONLY memory
SIGEXEC	28*	Execute from STACK/DATA space
SIGBND	29*	Segmentation violation
SIGUSR1	30	User-defined interrupt #1
SIGUSR2	31	User-defined interrupt #2
SIGUSR3	32	User-defined interrupt #3
SIGABORT	33	Program abort
SIGSPLR	34	Spooler interrupt
SIGINPUT	35	Input is ready
SIGDUMP	36	Memory dump
SIGMILLI	62	Millisecond alarm
SIGEVT	63	Mouse/keyboard event interrupt

DIAGNOSTICS

Issues an error if the interrupt specified is out of range.

create

USAGE

```
sys create,fname,perm
<file descriptor in D0>
```

DESCRIPTION

Create creates a new file with the access permissions specified in *perm.* The permissions are the same as in the *chprm* call, and are:

```
FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
```

If the file already exists, its length is truncated to zero (all data deleted) but the original permissions and owner is retained. In either case, the file is ultimately opened for writing. It is not necessary to specify write permission even though the file will ultimately be opened for writing. This allows a task to create a file and disallow others from writing the file until the task has been completed.

DIAGNOSTICS

Issues an error issued if too many files are open, if the files path can not be searched, or if the directory it resides in cannot be written.

create_pty

USAGE

sys create_pty
<slave file descriptor in D0>
<master file descriptor in A0>

DESCRIPTION

This function creates a new pseudo-terminal channel. The file descriptor for slave access is returned in fd[0]. The file descriptor for master access is returned in fd[1].

Pseudo-terminals must exist as real devices in the device directory named utility via the command:

```
        makdev /dev/ptyxx p 1 xx
```

where ´xx´ is a decimal number with a possible leading zero.

The function *create_pty* returns access to the first unused pseudo-terminal channel in the system. As these channels are closed, they will be reused in numerical order. I.e. *create_pty* will always return the lowest numbered pesudo-terminal channel not currently in use.

Once the channel has been opened using *create_pty*, additional slave accesses may be obtained using *open* for the appropriate device.

For slave access, this channel is exactly the same as a normal terminal. For master access, writing to the channel is seen as input on the slave side and reading from the channel reads characters output from the slave side.

The function *ofstat()* may be applied to a pseudo-terminal. The only difference from a normal terminal is that the the mode will be *S_SLAVE_PTY* or *S_MASTER_PTY*.

DIAGNOSTICS

Issues an error if no pseudo-terminal channels are available.

crpipe

USAGE

```
sys crpipe
<read file descriptor in D0>
<write file descriptor in A0>
```

DESCRIPTION

This call creates a pipe for inter-task communication. This call should be used before a *fork* operation, to allow the output of the original task to be used as input by the forked task. Up to 4096 bytes of output can be written into the pipe before the task is suspended. Once the task doing the reading has read all of the data written, the writing task run again. If the writing task closes the file (file descriptor from A0) and the reading task consumes all of the data, an end-of-file condition results.

DIAGNOSTICS

Issues an error if too many files and pipes are opened.

crtsd

USAGE

sys crtsd,fname,desc,address

DESCRIPTION

This call creates a special file (device) or a new directory. *Fname* specifies the name of the new file; *desc* is a 16-bit descriptor that describes the file's type and permissions. If the file being created is a special file, the *address* argument specifies the internal device number. The descriptor has the *type* as the most significant byte and the *permissions* as the least significant byte. Their definitions follow:

Types

TPBLK => %00000010 ($02) block type device
TPCHR => %00000100 ($04) character type device
TPDIR => %00001000 ($08) directory type file

Permissions

FACUR => %00000001 ($01) owner read permission
FACUW => %00000010 ($02) owner write permission
FACUE => %00000100 ($04) owner execute permission
FACOR => %00001000 ($08) others read permission
FACOW => %00010000 ($10) others write permission
FACOE => %00100000 ($20) others execute permission
FXSET => %01000000 ($40) set id bit for execute

DIAGNOSTICS

Issues an error if the file already exists or if the caller is not the system manager.

defacc

USAGE

sys defacc,perm

DESCRIPTION

Defacc set the default access permissions as specified by *perm*. Normally, when a file is created, it is given the permissions specified in the *create* system call. The value specified by *create* is ANDed with the one´s-compliment of a per-task value known as the default permissions. This process turns off or disables the permissions contained in the default permissions byte, no matter what the specified permissions are in the create call. The *defacc* call is used to set the default permissions. All *forks* and *execs* pass on the existing default value. See *chprm* for a list of the permission bits and their meaning.

DIAGNOSTICS

No errors generated.

dup

USAGE

```
<file descriptor in D0>
sys dup
<file descriptor in D0>
```

DESCRIPTION

Dup duplicates the specified file descriptor; in other words, the file associated with the file descriptor is opened again and given another descriptor, which is returned. The new file is opened with the same mode as the original (e.g., if the original was open for *read*, so will the new one).

DIAGNOSTICS

Issues an error if too many files are opened or the file descriptor is invalid.

dups

USAGE

```
<file descriptor in D0>
<specified descriptor in A0>
sys dups
<file descriptor in D0>
```

DESCRIPTION

This call is like *dup* except the caller may specify the file descriptor of the duplicated open file. If the specified descriptor is already open, it is closed before being duplicated.

DIAGNOSTICS

Issues an error if too many files are open, or if the file descriptors are invalid.

exec

USAGE

```
sys exec,fname,arglist
...
fname fcc " .... ",0
...
arglst fqb arg0,arg1,... ,0
arg0 fcc " .... ",0
arg1 fcc " .... ",0
```

DESCRIPTION

The *exec* system call executes a binary file. *Fname* specifies the file to be executed. The calling task will be terminated and the new one started up. There is no return from a successful *exec*. A return indicates an error condition. All open files remain open through the *exec*. Interrupts that are being ignored stay in that state, but those that are being caught are reset to their default state.

When the file starts executing, the following arguments are available:

```
... highest address in task space ...
0
...
arg0: <arg0 >
0
argn
...
arg0
sp -> argcnt
...  low memory ...
```

The stack pointer is pointing at a 4-byte argument count. Above that is a list of pointers to the actual arguments, which are at the highest part of memory. Two zero bytes are left at the very top of the task address space.

DIAGNOSTICS

Results in an error (and a return to the caller of exec) if the file does not exist, it was not executable binary, there were too many arguments (approximately 3,000 bytes max), or the memory space was exceeded.

exece

USAGE

```
sys exece,fname,arglist
...
fname fcc " . . . . ",0
...
arglst fqb arg0,arg1, . . . ,0
arg0 fcc " . . . . ",0
arg1 fcc " . . . . ",0
```

DESCRIPTION

The *exece* system call executes a binary file. *Fname* specifies the file to be executed. The calling task will be terminated and the new one started up. There is no return from a successful *exece*. A return indicates an error condition. All open files remain open through the *exece*. Interrupts that are being ignored stay in that state, but those that are being caught are reset to their default state.

When the file starts executing, the following arguments and environment variables are available:

```
... highest address in task space ...
0
...
env_var0: <env_var0>
0
...
arg0: <arg0 >
0
env_varn
...
env_varn0
0
argn
...
arg0
sp -> argcnt
...  low memory ...
```

Arguments are passed to a program by leaving them on the system stack. When initiating a program, the system stack pointer (A7) is left pointing at the argument count (see Figure 1). Any arguments passed to the program are found in a special format just above the argument count. The environment variables are also found in this area.

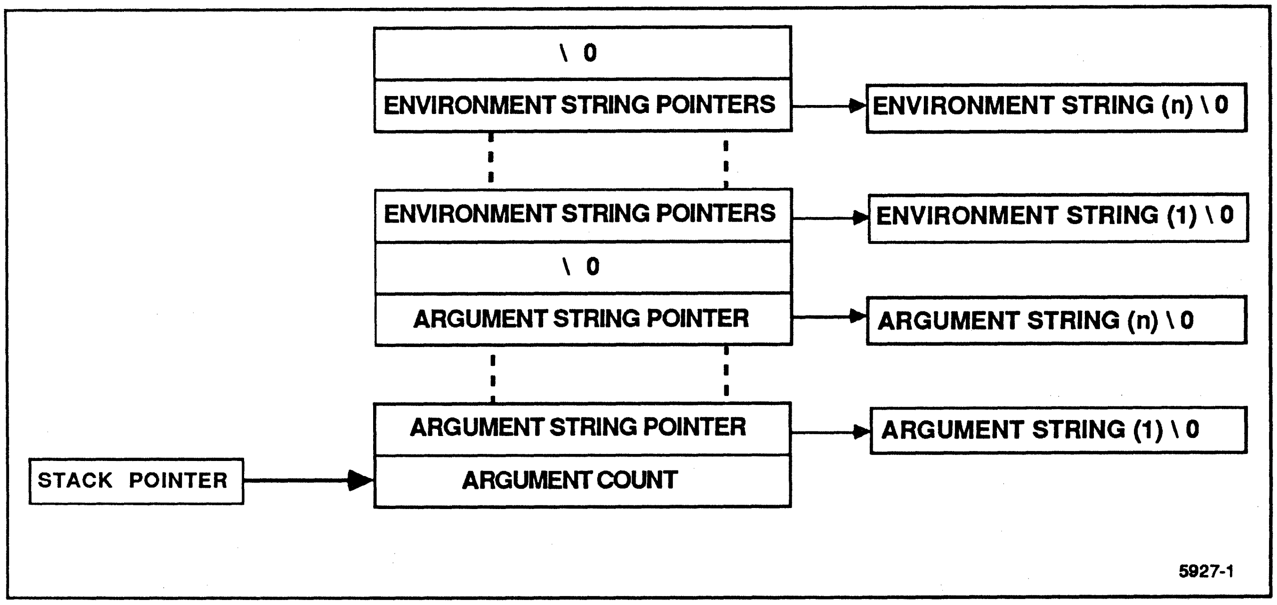

Figure 4-1. Argument And Environment Variables.

The arguments themselves are simply strings of characters which the program must know how to use. In order to easily find these strings, the system provides a list of pointers to the beginning of the strings. In addition, the system provides a count of how many arguments have been passed.

The pointers to the environment variables are found in memory, directly above the pointers to the arguments. Unlike argument strings, there is no count of the pointers to the environment variables, however they are terminated by a null pointer.

DIAGNOSTICS

Results in an error (and a return to the caller of exece) if the file does not exist, it was not executable binary, there were too many arguments (approximately 3,000 bytes max), or the memory space was exceeded.

fcntl

USAGE

<file-descriptor in D0>
sys fcntl,function
<state in D0>

DESCRIPTION

This function is used to change or interrogate the behaviour of a file in the system. Various behaviours may be modified on a file-by-file, task-by-task basis. Each behaviour may be set/reset by using a specific function to the *fcntl()* function.

The function returns a mask that indicating the state of the behaviours that can be modified.

Currently, the functions available are:

FCNTL_NOBLOCK	Subsequent read operations on this file descriptor will not cause the task to be suspended if no data is available. In this mode, the error ENOINPUT will be returned if no data is available and the signal "INPUT READY" will be sent to the task when data becomes available.
FCNTL_BLOCK	Returns the file descriptor to normal blocking mode.
FCNTL_GET_PARAMS	Returns the state mask.
FCNTL_INPUT_FD	Returns the file descriptor (in D0) of the last file which sent the signal "INPUT READY".

The value returned is a combination of the state bits below:

FCNTL_P_BLOCK	Reads from the file will not cause the task to be suspended. Also, the signal "INPUT READY" will be sent when input becomes available.

DIAGNOSTICS

No errors are reported

filtim

USAGE

```
<time in D0>
sys filtim,fname
```

DESCRIPTION

Filtim sets the *last modified time* of the specified file to the value contained in the D0 register. The operating system represents time as the number of seconds that has elased since the epoch. It defines the epoch as 00:00 (midnight) January 1, 1980 Grennwich Mean Time. Only the system manager can execute this call.

DIAGNOSTICS

Returns an error if the file does not exist, if the file is currently open by another task, or if the caller is not the system manager.

fork

USAGE

```
sys fork
<new task returns here>
<old task here (pc+2), new task id in D0>
```

DESCRIPTION

Fork creates a new task. The new task inherits a copy of the caller's core image, all open files, and file pointers. The new task is identical to the original except that the old task returns 2 bytes past the system call and has the newly created task's id in the D0 register.

DIAGNOSTICS

Issues an error if more than 32 tasks have been created or the system task table is full.

gtid

USAGE

```
sys gtid
<task id in D0>
```

DESCRIPTION

This call returns the current task's system id. This number can be used to generate unique file names.

DIAGNOSTICS

No errors are returned.

guid

USAGE

sys guid
<actual user id in D0>
<effective user id in A0>

DESCRIPTION

Guid returns both the actual user id (which identifies the login id of person logged on the system) and the effective id (which defines the current access permissions of the running task).

DIAGNOSTICS

No errors are possible.

ind

USAGE

sys ind,label

DESCRIPTION

The *ind* system call is used where it is necessary to create system calls or their arguments on the fly (in the running program). The *label* points to an address that contains the actual call and its arguments. The task resumes execution after the *sys ind* and not after the labeled code. Another *ind* or *indx* call may not be called from *ind*.

DIAGNOSTICS

Issues an error if the value at the *label* is not a valid system call, or if it is an indirect call.

indx

USAGE

sys indx

DESCRIPTION

This call is similar to *ind,* but allows the system function code and arguments to be anywhere in memory, including the stack. Where *ind* had a label pointing to the system call and parameters, this call requires A0 to point to the call and parameters. One application of *indx* is to push the arguments and system call code on the stack, point to the call, then issue an *indx* call. Another *ind* or *indx* call may not be called from *indx.*

DIAGNOSTICS

Reports an error if the system function is not a valid system call, or if it is another indirect call.

link

USAGE

```
sys link,fname1,fname2
```

DESCRIPTION

This call links *fname1* to *fname2*. After the link, reference to *fname2* will access the contents of *fname1*. The files contents and attributes are not changed in any way.

DIAGNOSTICS

Issues an error if *fname1* does not exist, if *fname2* already exists, if *fname2's* directory is write protected, if *fname1* is a directory, or if the file names are on different devices.

lock

USAGE

```
sys lock,flag
```

DESCRIPTION

Lock keeps a task from being swapped (that is, it locks a task in memory). Only the system manager may execute this call. If *flag* is non-zero, the task is locked; if it is zero, the task is unlocked.

DIAGNOSTICS

Issues an error if the caller is not the system manager.

lrec

USAGE

```
<file descriptor in D0>
sys lrec,count
```

DESCRIPTION

Lrec makes an entry in the system's locked record table. Before the new entry is made, all other entries in the table associated with the calling task and the specified file is removed. *Count* represents the number of bytes in the file (record size) to be locked from the current file position. If the specified record overlaps any part of another task's entry in the lock table for the same file, an error results (ELOCK). Only regular files may be referenced (e.g., no devices, pipes, or directories). Closing a file removes the lock table entry created as does the *urec* system call. Note that the part of the file specified is not actually *locked* from other's use, but proper use of the *lrec* and *urec* calls will have the same effect.

DIAGNOSTICS

Produces an error if there is no file for the specified descriptor, the file is not a regular file, the record is locked by another task, or the lock table is temporarily full.

memman

USAGE

sys memman,function,start_address,end_address

DESCRIPTION

The *memman* system call is used to control regions of memory. The region of a task's logical address space is specified by *start_address* and *end_address*.

In all cases, the region operated on is a set of pages and may be different than the address range specified. Depending upon the function selected, the pages selected are rounded outward or inward to the next page boundary. The function arguments, 0 - 5, defines control activity that rounds the page size outward. By adding 32 to the function argument, the page size is rounded inward.

The *function* argument defines the control activity:

Rounds Page Size	Function	Operation
Outward	0	Clear the "dirty bit"
	1	Lock the region in memory
	2	Unlock the region from memory
	3	Write disable the region
	4	Write enable the region
	5	Release the storage associated with the region
Inward	32+0	Clear the "dirty bit"
	32+1	Lock the region in memory
	32+2	Unlock the region from memory
	32+3	Write disable the region
	32+4	Write enable the region
	32+5	Release the storage associated with the region

DIAGNOSTICS

Issues an error if the function number is not valid, the address range specified is out of the task's address space, or if the *start_address* is greater the the *end_address*.

mount

USAGE

```
sys mount,sname,fname,mode
```

DESCRIPTION

Mount mounts a special file on the file system. The file *fname* should be a directory; after the mount, any reference to *fname* will reference the root directory of the special file (block device) *sname*. The *mode* is normally 0; if it is non-zero, the device is mounted as *read only* (i.e. writing not permitted).

DIAGNOSTICS

Issues an error if *sname* is not an appropriate file, if it is already mounted, if *fname* does not exist, or if too many devices are currently mounted.

ofstat

USAGE

```
<file descriptor in D0>
sys ofstat,buffer
```

DESCRIPTION

This call returns the status of an open file. The file is referenced by its file descriptor (obtained when the file was opened or created). The status information is returned in the user space pointed at by *buffer*. See the *status* call for a description of the returned information.

DIAGNOSTICS

Returns an error if the file descriptor is not valid (i.e. the file is not open or the descriptor is out-of-range).

open

USAGE

```
sys open,fname,mode
<file descriptor in D0>
```

DESCRIPTION

Open opens an existing file called *fname*. The file is opened for reading if *mode* is 0, for writing if *mode* is 1, or for both reading and writing if *mode* is 2. *Open* returns a file descriptor that must be used for future file references.

DIAGNOSTICS

An error will be issued if the file does not exist, the path directories cannot be searched, too many files are open, or the permissions do not grant the requested mode.

phys

USAGE

```
sys phys,object
<logical base address in D0>
```

DESCRIPTION

The *phys* system call permits access to certain system resources. Resources represented by *object* are:

Object	Resource
1	Display bit map
2	Reserved
3	Reserved
4	Time of day clock

The object numbers are defined above. If the number is positive, the resource is mapped into the task´s address space. If the number is negative, it is mapped out. An object number of 0 unmaps all previously mapped in resources. The logical address of the base of the mapped in resource is returned in D0.

DIAGNOSTICS

Returns an error if the object number is not valid.

profile

USAGE

```
sys profil,prpc,buffer,bsize,scale
```

DESCRIPTION

The *profile* call sets up a buffer and parameters that the system uses to profile a running task. If profiling is enabled, each time a clock tick occurs (every tenth second) a word in the *buffer* that corresponds to the current value of the program counter in the running task is incremented. The *prpc* value represents the lowest address in the running task to be profiled. The argument *buffer* specifies the address of the profile buffer, and *bsize* specifies its size. The buffer size also determines the highest address in the running task to be profiled since *pc* addresses too large to be mapped into the buffer are ignored.

The *scale* value is used to scale the task program counter and must be a power of 2 (maximum size is 128). Profiling may be disabled by setting *scale* to 0 or 1.

Here´s what happens when a clock interrupt occurs during execution of a task for which profiling is enabled:

1. The profile value *prpc* is subtracted from the task´s current program counter, and the result is divided by the scale factor.
2. This value is then multiplied by 2 to form an offset into the *buffer*.
3. If this offset is less than *bsize*, the 16 bit word residing at *buffer+offset* is incremented by one.

DIAGNOSTICS

No errors are issued.

read

USAGE

```
<file descriptor in D0>
sys read,buffer,count
<bytes read in D0>
```

DESCRIPTION

This call reads the file represented by the specified file descriptor. The memory in the user's space pointed to by *buffer* is filled with data from the file. A maximum of *count* bytes is read. All bytes requested will not necessarily be returned. If the file is a terminal, at most, one line is returned. If the returned byte count is zero and no error is reported, the end-of-file has been reached. I/O requests block the current program until the read is fullfilled.

DIAGNOSTICS

Issues an error if a physical I/O error occurred, or if a bad file descriptor or bad count was specified.

rump

USAGE

```
<resource-name in A0>
sys rump,function
```

DESCRIPTION

The function RUMP_CREATE creates a new named resource. The purpose of such resources is to provide a mechanism for controlling access to physical resources such as I/O devices or special shared memory. There are four operations which may be applied to a named resource. These are:

create	create the resource
destroy	remove the resource from the system
enqueue	obtain exclusive access to the resource
dequeue	relinquish access to the resource.

If the create function succeeds, a new named resource will be created with the name given by the argument *resource.* This must be a NULL terminated character string of 16 or fewer characters (including the NULL). Otherwise, it returns the system error code in D0.

The create function does not give access of the resource to to the creator.

The function *RUMP_ENQUEUE* obtains exclusive access to the named resource for the task.

If the function succeeds, access to the named resource with the name given by the argument *resource* will be granted. Otherwise, it returns the system error code in D0.

If any other task currently has access to the resource, the calling task will wait until the resource becomes free. This waiting is done in a First-In/First-Out fashion to guarantee equal access to all tasks.

The function RUMP_DEQUEUE releases access to the named resource.

If the function succeeds, access to the named resource with the name given by the argument *resource* will be given up. Otherwise, it returns the system error code in D0.

If any other tasks are currently waiting for access to the resource, then the first such task will be given access to the resource.

The function *RUMP_DESTROY* destroys a named resource.

If the function succeeds, the named resource with the name given by the argument *resource* will be destroyed. Otherwise, it returns the system error code in D0.

Only an *idle* resource can be destroyed. If any task currently has access to the resource, the destroy function is not permitted.

DIAGNOSTICS

No errors are reported

seek

USAGE

```
<file descriptor in D0>
sys seek,position,type
<position in D0>
```

DESCRIPTION

Seek positions a file's read/write pointer to the specified file location. The file is specified by the file descriptor. The argument *position* represents a four-byte, signed offset. The starting point for this offset is determined as follows by the *type* argument:

type	**starting position**
0	Position from the beginning of the file
1	Position from the current position
2	Position from the end of the file

The returned value is the resulting position of the file.

If a *seek* is performed past the end of the file when writing, a gap in the file is created (no actual device space is allocated). This gap is read as zeros. To determine the current position in the file, use *sys seek,0,1*, which positions the pointer 0 bytes from the current position.

DIAGNOSTICS

Returns an error if a file descriptor is invalid or if the *seek* is attempted on a pipe.

setpr

USAGE

```
<priority bias in D0>
sys setpr
```

DESCRIPTION

Setpr sets the priority bias used by the system scheduler. The value specified is subtracted from the normal user priority, so the effect is that of lowering the task's priority. Only the system manager may specify negative arguments (which will increase the task's priority). The priority bias specified should be in the range of 25 to -25.

DIAGNOSTICS

No errors are issued.

spint

USAGE

```
<task number in D0>
sys spint,interrupt
```

DESCRIPTION

This call sends a program interrupt to a task. The task is specified by its task number; the receiving task must have the same effective user id unless the caller is the system manager. The *interrupt* argument specifies which interrupt to send. See *cpint* for a list of interrupts.

If the specified task number is zero, the interrupt is sent to all tasks associated with the caller´s control terminal. If the task number is -1 and the caller is the system manager, the interrupt is sent to all tasks in the system with the exception of tasks 0 and 1 (the scheduler and the initializer).

DIAGNOSTICS

Issues an error if the specified task does not exist or if the effective user id´s do not match.

stack

USAGE

```
<address in A0>
sys stack
```

DESCRIPTION

The system extends the user's stack memory to include the address specified. If the address is higher than what is currently allocated, all lower memory is released to the system. A task initially starts with stack space between 100 and 3000 bytes depending on the number of arguments passed from exec. The system automatically allocates additional space to the stack. This call allows explicit control over the stock allocation.

DIAGNOSTICS

Issues an error if the request for memory overflows into the data segment.

status

USAGE

sys status,fname,buffer

DESCRIPTION

The file *fname* has its status read and returned to the user in the space specified by *buffer*. The data returned by this call (as well as *ofstat*) has the following format:

```
* buffer begin *

st_dev  rmb  2   device number
st_fdn  rmb  2   fdn number
st_fil  rmb  1   filler for alignment
st_mod  rmb  1   file modes - see below -
st_prm  rmb  1   permission bits - see below -
st_cnt  rmb  1   link count
st-own  rmb  2   file owner's user id
st_siz  rmb  4   file size in bytes
st_mtm  rmb  4   last time file was modified
st_spr  rmb  4   future use only

* mode codes

FSBLK => %00000010  ($2)  block device
FSCHR => %00000100  ($4)  character device
FSDIR => %00001000  ($8)  directory

* permissions

FACUR => %00000001 ($01)  owner read permission
FACUW => %00000010 ($02)  owner write permission
FACUE => %00000100 ($04)  owner execute permission
FACOR => %00001000 ($08)  others read permission
FACOW => %00010000 ($10)  others write permission
FACOE => %00100000 ($20)  others execute permission
FXSET => %01000000 ($40)  set id bit for execute
```

DIAGNOSTICS

Issues an error if the file does not exist or the directory path cannot be searched.

stime

USAGE

```
<time in D0>
sys stime
```

DESCRIPTION

This call sets the system time and date. The time is measured in seconds from 0000 January 1, 1980, local time. Only the system manager can execute this call.

DIAGNOSTICS

Reports an error if the caller is not the system manager.

stop

USAGE

```
sys stop
```

DESCRIPTION

Stop halts a task until a program interrupt is received from *spint* or *alarm*. When *stop* returns, it will always have an error (EINTR). The system error list is found in the file */lib/syserror*.

DIAGNOSTICS

Always returns with an error (EINTR).

suid

USAGE

```
<user id in D0>
sys suid
```

DESCRIPTION

This call sets the effective and actual user id. While a program is running, you can change the effective user id of the program´s user to the actual id of the program´s owner. This call can be executed only if the actual user id matches the id in the argument, or if the caller is the system manager. If you are the system manager, you can change the actual user id of any user.

DIAGNOSTICS

Issues an error if the caller is not the system manager or if the actual user id does not match.

term

USAGE

```
<status in D0>
sys term
```

DESCRIPTION

Term terminates the current task. The status specified is made available to the parent task. The status is usually zero if there were no errors in the terminating task. A non-zero status should indicate some error condition. This system call does not return to the caller.

DIAGNOSTICS

No errors reported.

time

USAGE

```
sys time,tbuf
```

DESCRIPTION

The *time* call returns the system´s current time. Internally, the time is kept as a four-byte number, representing the number of seconds that have elapsed since 0000 January 1, 1980 local time. The time information is placed at the address specified by *tbuf* and has the following format:

```
tm_sec  rmb  4  Time in seconds
tm_tik  rmb  1  Ticks in current second (tenths)
tm_dst  rmb  1  Reserved
tm_zon  rmb  2  Reserved
```

The *tm_tik* value may be used for finer measurements.

NOTE

The time *system call does not permit the result buffer to reside in the text segment. An attempt to do so results in an address error exception. Put the buffer in the data or stack segment.*

DIAGNOSTICS

No errors are issued.

truncate

USAGE

```
<file descriptor in D0>
sys truncate
```

DESCRIPTION

The truncate system call truncates an existing file´s size. The file must be opened for write and the file descriptor passed in D0. The file is truncated at the current file position. To truncate at a specified location, it is necessary to use the *seek* system call prior to truncate.

DIAGNOSTICS

Returns an error if the file descriptor is not valid or the file is not open for write.

ttime

USAGE

```
sys ttime,buffer
```

DESCRIPTION

This call is used to obtain the accounting time information about a task. All times are represented in tenths of seconds. The information is returned to the user at *buffer* and has the following format:

```
ti_usr rmb 4  Task's user time
ti_sys rmb 4  Task's system time
ti_chu rmb 4  Children's user time
ti_chs rmb 4  Children's system time
```

The child times shown are the totals of all children tasks spawned by this task and its children.

DIAGNOSTICS

No errors are issued.

ttyget

USAGE

```
<file descriptor in D0>
sys ttyget,ttbuf
```

DESCRIPTION

This call returns information about a terminal. The information returned is put in the 6-byte buffer pointed to by *ttbuf.* The following formats describe the data:

```
* ttbuf

tt_flg  rmb  1  Flags byte - see below -
tt_dly  rmb  1  Delay byte - reserved -
tt_cnc  rmb  1  Line cancel char (default is ^X)
tt_bks  rmb  1  Backspace character (default is ^H)
tt_spd  rmb  1  Terminal speed - see below -
tt_spr  rmb  1  Stop output byte - see below -

* flags

RAW    => %00000001 ($01)  Raw i/o mode
ECHO   => %00000010 ($02)  Echo input characters
XTABS  => %00000100 ($04)  Expand tabs on output
CRMOD  => %00010000 ($10)  Output cr and lf for cr
BSECH  => %00100000 ($20)  Echo backspace echo char
SCHR   => %01000000 ($40)  Single character input mode

* speeds

INCHR  => %10000000 ($80)  Input ready to be consumed

* stop output

XANY   => %00100000 ($20)  Accept any character to restart output
XONXOF => %01000000 ($40)  Enable XON/XOFF for start/stop output
ESCOFF => %10000000 ($80)  Disable ESC for start/stop output
```

For more information about the *ttyset* function, refer to the topic *THE "ttyset" AND "ttyget" FUNCTIONS* in section 2 of this manual.

DIAGNOSTICS

Returns an error if the specified file is not a character device.

ttynum

USAGE

sys ttynum
<terminal number in D0>

DESCRIPTION

This call returns the number of the calling task's terminal. For example, *tty02* returns $0002 in the D0 register.

DIAGNOSTICS

No errors are issued.

ttyset

USAGE

```
<file descriptor in D0>
sys ttyset,ttbuf
```

DESCRIPTION

This call sets device dependent information described in *ttyget*. For */dev/console*, the data in *ttbuf* is exactly as described in *ttyget*. For more information about the *ttyset* function, refer to the topic *THE "ttyset" AND "ttyget" FUNCTIONS* in section 2 of this manual.

In normal use, you would first execute a *ttyget* system call to obtain the existing configuration. Next, use the logical operators AND or OR to set or clear the desired bits. (Be careful not to alter any bits other than those that must be changed.) Finally, execute the *ttyset* system call.

DIAGNOSTICS

Issues an error if the file specified is not a character device.

unlink

USAGE

sys unlink,fname

DESCRIPTION

Unlink removes the *fname* entry from a directory. If this is the last link to the file, the file will be deleted and its device space will be freed. If the file is open, it will not be destroyed until the file is closed.

DIAGNOSTICS

Issues an error if the file does not exist, the directory cannot be written, or the directory path cannot be searched.

unmnt

USAGE

```
sys unmnt,sname
```

DESCRIPTION

This call unmounts a special file *sname* from the system. The file associated with the special file reverts to its ordinary interpretation (see mount).

DIAGNOSTICS

Issues an error if the file system specified is busy or is not mounted.

update

USAGE

```
sys update
```

DESCRIPTION

Update updates all information on the disks; it writes out all data that is in memory waiting to be written to the disks.

DIAGNOSTICS

No errors are reported.

urec

USAGE

```
<file descriptor in D0>
sys urec
```

DESCRIPTION

Urec removes an entry in the system's lock table (previously installed by *lrec*). All entries associated with the calling task and specified file are removed.

DIAGNOSTICS

Issues an error if the specified file descriptor is bad.

vfork

USAGE

```
sys vfork
<new task returns here>
<old task here (pc+2), new id in D0>
```

DESCRIPTION

Vfork is a more efficient *fork* operation and is only available on virtual memory systems. Its operation is identical to fork but instead of the child task receiving new memory, it uses the same memory as the parent. After a *vfork*, the parent is halted until the child task either terminates or execs another file.

There are several restrictions placed on the child task created by *vfork*. The system will not let the child change its memory size or execute the system calls memman, *fork*, or *vfork*. The user of *vfork* should make sure the child task does not alter the stack frame in any way or change data that the parent is not expecting changed.

DIAGNOSTICS

Issues an error if too many tasks have been created or if the system task table is full.

wait

USAGE

sys wait
<task id in D0>
<term status in A0>

DESCRIPTION

This call is used to wait for a program interrupt or the termination of a child task. A *wait* must be executed for each of a task´s children. The task id of the terminated task is returned, as well as its termination status. The low byte of this status is the value passed by the *term* system call. A non-zero value here usually represents some sort of error condition. The high byte of the status is zero for normal termination. If non-zero, this byte contains the interrupt number that caused it to terminate. If the most significant bit of the status is set, a core dump was produced as a result of termination. Consult *cpint* for a list of interrupt numbers.

DIAGNOSTICS

Issues an error if there are no children tasks.

write

USAGE

```
<file descriptor in D0>
sys write,buffer,count
<byte count written in D0>
```

Write writes *count* bytes of data from location *buffer* to the file specified by the file descriptor. If the returned byte count does not equal the requested count, it should be considered an error. Writes that are multiples of 512 bytes and begin on 512 byte address boundaries are the most efficient.

DIAGNOSTICS

Issues an error if the file descriptor is invalid or if a physical I/O error resulted.

Section 5
DISPLAY ACCESS FUNCTIONS

The operating system provides access to display functions through the processor's trap instruction. To invoke these functions, load register D0 with the function code (parameters for the functions go in other registers) and issue a trap #13 instruction.

On return, the carry bit is cleared if there were no errors. If an error occurs, the carry bit is set and register D0 contains an error code. Table 5-1, *Display Function Codes* summarizes the display function codes.

When functions pass an X,Y coordinate pair in registers, the X-coordinate is a signed 16-bit integer value in the upper half of the register, the Y-coordinate is a signed 16-bit integer value in the lower half of the register.

A program using two display functions follows. The program inverts the display so text is white on a black background, waits for two seconds, then returns the display to normal.

```
        lib     sysdef
        lib     sysints
        lib     sysdisplay
        text
start   sys     cpint,SIGALRM,wake  catch alarm and goto wake

        move.l  #whiteOnBlack,d0    whiteOnBlack equ #13
        trap    #13                 access display function

        move.l  #2,d0               set seconds for alarm
        sys     alarm               begin alarm call
        sys     stop                wait for alarm interrupt

wake    move.l  #blackOnWhite,d0    blackOnWhite equ #12
        trap    #13                 access display function

        move.l  #0,d0               get status in D0
        sys     term                terminate task
        end     start
```

Table 5-1
Display Function Codes

Code	Name	Description
0	curserOn	Displays the cursor
1	curserOff	Suspends display of the cursor
2	curserLink	Causes the cursor to track the mouse
3	curserUnlink	Breaks the links that caused the cursor to track the mouse
4	curserPanOn	Causes the viewport to pan when the cursor reaches an edge
5	cuserPanOff	Disables viewport panning via cursor movement
6	displayOn	Makes the display visible
7	displayOff	Blanks the display
8	joyPanOn	Turns on panning via joydisk
9	joyPanOff	Disconnects the joydisk from viewport panning
10	timeoutOn	Causes the screen to automatically blank if inactive for ten minutes
11	timeoutOff	Disables automatic blanking
12	blackOnWhite	Sets the display to Normal Video mode
13	whiteOnBlack	Sets the display to Inverse Video mode
14	terminalOn	Enables use of the terminal emulator with the display
15	terminalOff	Disables use of the terminal emulator with the display
16	getMousePoint	The position of the mouse is returned as an (X,Y) pair in the high and low halves of register D0
17	setMousePoint	The current mouse position is set to the position passed as an (X,Y) pair in the high and low halves of register D0
18	getCursorPoint	The current cursor position is returned as an (X,Y) pair in the high and low halves of register D0
19	setCursorPoint	The current cursor position is set to the position passed as an (X,Y) pair in register D0
20	getButtons	The state of the mouse buttons is returned in register D0
21	setSource	The source rectangle for a bitBlt operation is passes as an argument in registers D1 and D2
22	setDest	The destination rectangle for a bitBlt operation is passed as an argument in registers D1 and D2
23	updateComplete	This function allows the cursor to be displayed in areas previously specified as source or destination rectangles
24	getCursorform	This function gets the cursor
25	setCursorform	This stores the image of the cursor
26	getViewport	Returns the position of the upper left corner of the physical 640 X 480 physical viewport in the 1024 X 1024 virtual display
27	setViewport	Sets the display hardware to start updating from a specific position within the display bit-map
28	getDisplayState	The current state of the display is returned in a record pointed to by register A0
29	setKeyboardCode	The form of output generated by the keyboard is set by the value passed in D1
30	getMouseBounds	Return the limits of the rectangle within which the mouse and cursor are constrained in D0 and D
31	setMouseBounds	Set the limits of the rectangle within which the mouse and cursor are constrained
32	XYtoRC	Convert screen coordinates to terminal row and column
33	RCtoXY	Convert terminal row and column to screen coordinates
34	setCursorOffset	The cursor offset is passed as an (X,Y) pair in register D1
35	getCursorOffset	The cursor offset is returned as an (X,Y) pair in register D0

Display Functions

cursorOn

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 0

Displays the cursor. Returns 1 in D0 if the cursor was previously enabled, 0 if it was not.

cursorOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 1

Suspends display of the cursor. Returns 1 in D0 of the cursor was previously enabled, 0 if it was disabled.

cursorLink

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 2

Causes the cursor to track the mouse. The mouse location is set to the present cursor location. Returns 1 in D0 if the cursor was previously linked, 0 if it was not.

cursorUnlink

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 3

Breaks the links that caused the cursor to track the mouse. Returns 1 in D0 if the cursor and mouse were linked, 0 if not.

cursorPanOn

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 4

Causes the viewport to pan when the cursor reaches an edge. Returns 1 in D0 if previously enabled, 0 if not.

cursorPanOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 5

Disables viewport panning via cursor movement. Returns 1 in D0 if previously enabled, 0 if not.

displayOn

Usage

<display function in D0>
trap #13
<error code in D0>

Description

Display Function 6

Makes the display visible. Returns 1 in D0 if previously visible, 0 if blanked.

displayOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 7

Blanks the display, by turning the display off. Returns 1 in D0 if previously visible, 0 if blanked.

joyPanOn

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 8

Turns on panning via joydisk. Returns 1 in D0 if previously enabled, 0 if not.

joyPanOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 9

Disconnects the joydisk from viewport panning. Returns 1 in D0 if panning were previously enabled, 0 if not.

timeoutOn

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 10

Causes the screen to automatically blank if inactive for ten minutes. Returns 1 in D0 if previously enabled, 0 if not.

timeoutOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 11

Disables automatic blanking. Returns 1 in D0 if previously enabled, 0 if not.

blackOnWhite

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 12

Sets the display to Normal Video mode. Returns 1 in D0 if was previously black on white, 0 if white on black.

whiteOnBlack

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 13

Sets the display to Inverse Video mode. Returns 1 in D0 if was previously black on white, 0 if white on black.

terminalOn

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 14

Enables use of the terminal emulator with the display. Returns 1 in D0 if previously enabled, 0 if not.

terminalOff

Usage

<display function in D0>
trap #13
<if carry is cleared, previous state in D0>
<if carry is set, error code in D0>

Description

Display Function 15

Disables use of the terminal emulator with the display. Returns 1 in D0 if previously enabled, 0 if not.

getMousePoint

Usage

<display function in D0>
trap #13
<if carry is cleared, coordinates state in D0>
<if carry is set, error code in D0>

Description

Display Function 16

The position of the mouse is returned as an (X,Y) pair in the register D0. The X-coordinate is in the high half and the Y-coordinate is in the low half of register D0. If the cursor is linked to the mouse, this is the same as the mouse position.

setMousePoint

Usage

<display function in D0>
<new mouse X,Y pair in D1>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Display Function 17

The current mouse position is set to the position passed as an (X,Y) pair in the high (X) and low (Y) halves of register D0. If the cursor is linked to the mouse, the cursor position is also set.

getCursorPoint

Usage

<display function in D0>
trap #13
<if carry is cleared, coordinates state in D0>
<if carry is set, error code in D0>

Description

Display Function 18

The current cursor position is returned as an (X,Y) pair in the high (X) and low (Y) halves of register D0. If the cursor is linked to the mouse, this is the same as the mouse position.

setCursorPoint

Usage

<display function in D0>
<new cursor position X,Y pair in D1>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Display Function 19

The current cursor position is set to the position passed as an (X,Y) pair in register D1. The X-coordinate is in the top half of register D1 and the Y-coordinate is in the lower half of D1. If the mouse is linked to the cursor, the mouse position is also set.

getButtons

Usage

<display function in D0>
trap #13
<if carry is cleared, bitmask with current state of mouse buttons in D0>
<if carry is set, error code in D0>

Description

Display Function 20

The state of the mouse buttons is returned in register D0. Bit 0 corresponds to the right button, bit 1 to the middle button, and bit 2 to the left button. Zero in a bit indicates that the corresponding button is up, one indicates that it is pressed. The carry bit is cleared if there are no errors.

setSource

Usage

<display function in D0>
<X,Y coordinates of upper left corner of rectangle in D1>
<X,Y coordinates of lower right corner of rectangle in D2>
trap #13
<if carry cleared, undefined code in D0>
<if carry cleared, undefined code in D1>
<if carry cleared, undefined code in D2>
<if carry set, error code in D0>

Description

Display Function 21

The source rectangle for a bitBlt operation is passes as an argument in registers D1 and D2. It is encoded as:

```
upper-left-corner   (X0,Y0) in the high and low halves of
                      register D1

lower-right-corner  (X1,Y1) in the high and low halves of
                      register D2.
```

The operating system insures that the cursor is not displayed in this area. The carry bit is cleared to indicate no errors.

setDest

Usage

<display function in D0>
<X,Y coordinates of upper left corner of rectangle in D1>
<X,Y coordinates of lower right corner of rectangle in D2>
trap #13
<if carry cleared, undefined code in D0>
<if carry cleared, undefined code in D1>
<if carry cleared, undefined code in D2>
<if carry set, error code in D0>

Description

Display Function 22

The destination rectangle for a bitBlt operation is passed as an argument in registers D1 and D2. It is encoded as:

```
upper-left-corner  (X0,Y0) in the high and low halves of
                     register D1

lower-right-corner (X1,Y1) in the high and low halves of
                     register D2.
```

The operating system insures that the cursor is not displayed in this area. The carry bit is cleared to indicate no errors.

updateComplete

Usage

<display function in D0>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Display Function 23

This function allows the cursor to be displayed in areas previously specified as source or destination rectangles.

getCursorform

Usage

<display function in D0>
<address of cursor image in A0>
trap #13
<error code in D0>
<no change in A0>

Description

Display Function 24

This function gets the cursor (a 16 X 16 pixel bit-map stored as sixteen consecutive words). You must pass a pointer to this bit-map in register A0. The current cusor image is copied into the user's buffer

setCursorform

Usage

<display function in D0>
<address of cusor image in A0>
trap #13
<error code in D0>
<no change in A0>

Description

Display Function 25

This stores the image of the cursor (as a 16 X 16 bit-map of sixteen consecutive words) beginning at the address passed as a pointer in register A0. This changes the image of the graphics cursor. If the cursor is visible on the screen, the old image will immediatly be replaced by the new cursor.

getViewport

Usage

<display function in D0>
trap #13
<X,Y coordinate of the vewport's upper left corner in D0>

Description

Display Function 26

Returns the position of the upper left corner of the physical 640 X 480 physical viewport in the virtual display. This position is returned as an (X,Y) pair in the high and low halves of register D0. Refer to the appendices in the *4400 Series Operating System Reference* for information about the virtual display.

setViewport

Usage

<display function in D0>
<new X,Y position of viewport in D1>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error in D0>

Description

Display Function 27

Sets the display hardware to start updating from a specific position within the display bit-map. The position is specified as an X,Y pair passed in the high and low halves of register D1. This is used to position the viewport anywhere within the virtual display. Refer to the appendices in the *4400 Series Operating System Reference* for information about the virtual display. The 4406 accepts this command and returns with no changes.

getDisplayState

Usage

<display function in D0>
<address of buffer to receive state report in A0>
trap #13
<0 or error code in D0>
<no change in A0>

Description

Display Function 28

The current state of the display is returned in a record pointed to by register A0. The display state area must be at least 36 words (18-long) in length (at an even address). The structure is found in */lib/include/graphics.h*. The display state area contains the following information:

```
/* structure and bit definitions for save/restore state */
struct DISPSTATE{
        long statebits;      /*bits defines below*/
        struct POINT viewp; /*upper left corner of viewport*/
        struct POINT ulmouseb; /*upper left corner of mousebounds*/
        struct POINT lrmouseb; /*lower left corner of mousebounds*/
        short curarray[16]; /*the bits for the cursor */
        char keycode; /*kb encoding (0=event,1asni)*/
        char activefont; /*active font*/
        short lineincr;      /*byte increment between lines of screen*/
        short dispwidth; /*width of virtual display bitmap*/
        short dispheight; /*height of virtual display bitmap*/
        short viewwidth; /*width of visible viewport*/
        short viewheight; /*height of visible viewport*/
        short cursorxoffset; /*x offset for cursor*/
        short cursoryoffset; /*y offset for cursor*/
        long 11_reserved[2]; /*reserved for future use*/
};

#define DS_DISPON  0x0001 /*1-display enabled, 0=disabled*/
#define DS_SCRSAVE 0x0002 /*1=screen save enabled, 0=disabled*/
#define DS_VIDEO   0x0004 /*1=video normal,0=video inverse*/
#define DS_TERMEM  0x0008 /*1=terminal emulator enabled,0=disabled*/
#define DB_CAPSLOCK 0x0010 /*1=caps lock LED on, 0=off*/
                          /* 0x0020 reserved */
                          /* 0x0040 resreved */
                          /* 0x0080 reserved */
#define DS_CURSOR  0x0100 /*1=cursor enabled,0=disabled*/
#define DS_TRACK   0x0200 /*1=cursor tracks mouse,0=disabled*/
#define DS_PANCUR  0x0400 /*1=cursor panning enabled,0=disabled*/
#define DS_PANDISK 0x0800 /*1=joydisk panning enabled,0=disabled*/
                          /* 0x1000 reserved */
                          /* 0x2000 reserved */
                          /* 0x4000 reserved */
                          /* 0x8000 reserved */
#define DS_KBEVENTS 0x10000 /*1=keyboard generates event codes,0=not*/
                          /* 0x20000 through 0x80000000 are reserved */
```

setKeyboardCode

Usage

<display function in D0>
<code set designator in D1>
trap #13
<carry clear - old code set designator in D0>
<carry set - error code in D0>

Description

Display Function 29

The form of output generated by the keyboard is set by the value passed in D1. Valid values are:

```
0   sets keyboard output to event codes

1   sets keyboard output to ANSI terminal code sequences.
```

This call is normally only used after an *Enable Event Processing* call which implicitly sets the keyboard code to 0 (event codes). The previous keyboard code is return in D0.

getMouseBounds

Usage

<display function in D0>
trap #13
<if carry cleared, X,Y coordinate of upper left corner mouse bounds in D0>
<if carry cleared, X,Y coordinate of lower right corner mouse bounds in D0>
<if carry set, error code in D0>

Description

Display Function 30

Return the limits of the rectangle within which the mouse and cursor are constrained in D0 and D1. D0 contains the coordinates of the upper left corner of the rectangle. D1 contains the coordinates of the lower right corner. The upper half of each register is the X-coordinate, the lower half is the Y-coordinate.

setMouseBounds

Usage

<display function in D0>
<new X,Y pair upper left corner mouse bound in D1>
<new X,Y pair lower right corner mouse bound in D2>
trap #13
<if carry cleard 0 in D0>
<if carry set error code in D0>

Description

Display Function 31

Set the limits of the rectangle within which the mouse and cursor are constrained. D1 contains the coordinates of the upper left corner of the rectangle. D2 contains the coordinates of the lower right corner. The upper half of each register is the X-coordinate, the lower half is the Y-coordinate.

XYtoRC

Usage

<display function in D0>
<X,Y coordinate in D1>
trap #13
<if carry cleared, character row, column containing point in D0>
<if carry set, error code in D0>

Description

Display Function 32

Convert screen coordinates to terminal row and column. D1 contains the coordinates of a point on the portion of the virtual display used by the ANSI terminal emulator. Upon return the top half of D0 will contain the index of the terminal character row which contains that point. The lower half of D0 will contain the index of the character column.

RCtoXY

Usage

<display function in D0>
<character row, column in D1>
trap #13
<if carry cleared, X,Y coordinate of upper left corner character in D0>
<if carry cleared, width and height of character in D1>
<if carry set, error code in D0>

Description

Display Function 33

Convert terminal row and column to screen coordinates. The top half of D1 contains the index of a terminal character row and the lower half of D1 contains the index of the character column. Upon return D0 contains the coordinate of the upper left corner of the character cell. The top half of D1 contains the width (in pixels) of the character cell and the bottom half contains the height of the character cell.

setCursorOffset

Usage

<display function in D0>
<X,Y offset of cursor-point in D1>
trap #13
<error code in D0>

Description

Display Function 34

The cursor-point offset is passed as an (X,Y) pair in register D1. This offset is the distance from the top-left corner (0,0) of the cursor description to the point that is used as the reference for position and control of the cursor.

getCursorOffset

Usage

<display function in D0>
trap #13
<if carry is cleared, X,Y offset of the cursor-point in D0>
<if carry is set, error code in D0>

Description

Display Function 35

The cursor-point offset is returned as an (X,Y) pair in register D0. The offset is the distance from the top-left corner (0,0) of the cursor description to the point that is used as the reference for position and control of the cursor.

Section 6

KEYBOARD AND MOUSE FUNCTIONS

THE EVENT MANAGER

The event manager creates a buffered stream of 16-bit values which encode actual events. In general, the high-order 4 bits of the values are event type codes and the low-order 12 bits are event parameters. The following event-type codes are assigned:

0	delta time
1	mouse X location
2	mouse Y location
3	key or button pressed
4	key or button released
5	absolute time

Whenever the keyboard or mouse changes state, a time event is generated (either a type 0 or type 5 event) which reports the time of the event. This is followed by an event value which specifies the actual change which occurred.

Event Manager Functions

The operating system provides access to the event manager functions through the same mechanism as to the display functions (see section 5, *Display Access Functions*). The trap #13 instruction invokes the function whose code is passed in register D0 (see section 2, topic *Hardware Access Traps*). The event manager functions are summarized in Table 6-1, *Event manager functions* and described in the text following.

Table 6-1
Event manager functions

Code	Name	Description
40	eventsEnable	Turns on the event manager
41	eventsDisable	Turns off the event manager
42	eventSignalOn	Requests notification when events occur
43	eventMouseInterval	Specifies how often mouse motion events are created for a continuously moving mouse
44	getEventCount	Returns in register D0 the number of event values in the event buffer waiting to be processed
45	getNewEventCount	Returns in register D0 the number of event values in the event buffer which have occurred since the previous call to this function
46	getNextEvent	Returns in register D0 the next event value in the event buffer
47	getMillisecondTime	Returns in register D0 the number of milliseconds since the system was turned on (a 32-bit value)
48	setAlarmTime	A 32-bit millisecond time relative to system power-up in register D0
49	clearAlarm	Clears any pending alarms that the process has requested

eventsEnable

Usage

<event funtion in D0>
trap #13
<if carry is cleared, previous state of events in D0>
<if carry is set, error code in D0>

Description

Event Function 40

Turns on the event manager. Any subsequent user input action will cause event values to be created. Normal keyboard input through the *console* device and terminal emulator are disabled. Register D0 contains 0 if events were disabled, non-zero if events were disabled, before the call.

eventsDisable

Usage

<event funtion in D0>
trap #13
<if carry is cleared, previous state of events in D0>
<if carry is set, error code in D0>

Description

Event Function 41

Turns off the event manager. Keyboard input through the *console* device and terminal emulator is enabled. Register D0 contains 0 if events were disabled, non-zero if events were disabled, before the call.

eventSignalOn

Usage

<event funtion in D0>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Event Function 42

Requests the event manager to signal the current process when events occur. The event signal is disabled after being issued.

eventMouseInterval

Usage

<event funtion in D0>
<frequency to create mouse motion events in D1>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Event Function 43

Specifies how frequently mouse motion events are to be created if the mouse is continuously moving. The frequency value is passed in register D1 and is specified in units of milliseconds (granularity of milliseconds). A value of 0 indicates that mouse motion events should not be created

getEventCount

Usage

<event funtion in D0>
trap #13
<if carry is cleared, number of entries in event queue3 is in D0>
<if carry is set, error code in D0>

Description

Event Function 44

Returns in register D0 the number of event values in the event buffer waiting to be processed.

getNewEventCount

Usage

<event funtion in D0>
trap #13
<if carry is cleared, number of new entries since last call is in D0>
<if carry is set, error code in D0>

Description

Event Function 45

Returns in register D0 the number of event values in the event buffer which have occurred since the previous call to this function.

getNextEvent

Usage

<event funtion in D0>
trap #13
<if carry is cleared, the next event entry in D0>
<if carry is set, error code in D0>

Description

Event Function 46

Returns in register D0, the next event value in the event buffer.

getMillisecondTime

Usage

<event funtion in D0>
trap #13
<if carry is cleared, current millisecond clock value in D0>
<if carry is set, error code in D0>

Description

Event Function 47

Returns in register D0 the number of milliseconds since the system was turned on (a 32-bit value).

setAlarmTime

Usage

<event funtion in D0>
<time at which to signal in D1>
trap #13
<if carry is cleared, 0 in D0>
<if carry is set, error code in D0>

Description

Event Function 48

A 32-bit millisecond time relative to system power-up is passed in register D0. The requesting process will be signaled when this time is reached.

clearAlarm

Usage

<event funtion in D0>
trap #13
<if carry is cleared,current pending alarm time, or 0, in D0>
<if carry is set, error code in D0>

Description

Event Function 49

Clears any pending alarms that the process has requested. Returns the millisecond value of any pending alarms, or zero if there are no pending alarms, in D0.

Event Manager Key Codes

Each key on the keyboard, each position of the joydisk, and each of the mouse buttons has an event driver code associated with it. Table 6-2 shows the event code associated with each key.

Table 6-2
Keys and Event Driver Codes

Key Label	Event Code	Key Label	Event Code	Key Label	Event Code
Backspace	8	F	102	Break	141
Tab	9	G	103	Enter	150
Line Feed	10	H	104	Pad ,	151
Return	13	I	105	Pad -	152
Escape	27	J	106	Pad .	153
(space bar)	32	K	107	Pad 0	154
' "	39	L	108	Pad 1	155
> .	46	M	109	Pad 2	156
< ,	44	N	110	Pad 3	157
- _	45	O	111	Pad 4	158
/ ?	47	P	112	Pad 5	159
0)	48	Q	113	Pad 6	160
1 !	49	R	114	Pad 7	161
2 @	50	S	115	Pad 8	162
3 #	51	T	116	Pad 9	163
4 $	52	U	117	F1	201
5 %	53	V	118	F2	202
6 ^	54	W	119	F3	203
7 &	55	X	120	F4	204
8 *	56	Y	121	F5	205
9 (	57	Z	122	F6	206
; :	59	| ~	124	F7	207
= +	61	Rubout	127	F8	208
[{	91	Mouse right	128	F9	209
\ ‘	92	Mouse middle	129	F10	210
] }	93	Mouse left	130	F11	211
A	97	Shift (left)	136	F12	212
B	98	Shift (right)	137	Joydisk up	213
C	99	Control	138	Joydisk down	214
D	100	Caps lock	139	Joydisk right	215
E	101	← ↑	140	Joydisk left	216

Section 7
FLOATING POINT SUPPORT

The operating system provides access to the floating point hardware. Floating point values are in IEEE format. Both 32-bit single precision and 64-bit double precision formats are supported.

These operations are invoked by a trap #12 instruction with function code and arguments stored in registers. The floating point function code is passed in register D2. Operands are passed in registers D0 and A0 if they are single precision or integer, or in register pairs D0/D1 and A0/A1 if they are double precision. If only one operand is required it is passed in D0 (or D0/D1). The result is returned in register D0 for single precision, and in register pair D0/D1 for double precision.

For subtracts, compares, and divides, the value in register A0 (or A0/A1) is subtracted from, compared to, and divided into the value in register D0 (or D0/D1). For compare operations the processor's condition codes are set to reflect the result of the comparison. The floor function converts a floating point number to the largest integer less than or equal to it. The file */lib/sysfloat* contains symbolic definitions of the floating point functions for use by assembly language programs and are summarized in Table 7-1, *Floating Point Function Codes*. For compatibility the MC68020/68881 based systems of the 4400 series products emulate this interface.

Table 7-1
Floating Point Function Codes

Code	Name	Description
0	FADD	Add two single precision numbers
1	FSUB	Subtract two single precision numbers
2	FMUL	Multiply two single precision numbers
3	FDIV	Divide two single precision numbers
4	FCMP	Compare two single precision numbers
5	FNEG	Negate a single precision number
6	FABS	Take absolute value of a single precision number
7	FItoF	Convert integer to single precision floating point
8	FFtoIr	Round single precision floating point to integer
9	FTtoIt	Truncate single precision floating point to integer
10	FFtoIt	Floor function for single precision numbers
11	FFtoD	Convert single precision number to double precision
12	FDtoF	Convert double precision number to single precision
13	FDADD	Add two double precision numbers
14	FDSUB	Subtract two double precision numbers
15	FDMUL	Multiply two double precision numbers
16	FDDIV	Divide two double precision numbers
17	FDCMP	Compare two double precision numbers
18	FDNEG	Negate a double precision number
19	FDABS	Take absolute value of a double precision number
20	FItoD	Convert an integer to double precision floating point
21	FDtoIr	Round double precision floating point to integer
22	FDtoIt	Truncate double precision floating point to integer
23	FDtoIf	Floor function for double precision numbers
24	FsetStat	The value in D0 is written into the 32081's Status Register
25	FgetStat	The value of the 32081's status register is returned in D0

Floating Point Returns

A successfull execution of a floating point system call returns with the V (overflow) bit cleared. If an error occurs, the routine returns with the V bit set and the error indicated by the contents of Register D0. The error codes are:

Result had an underflow. *Trap on underflow* was enabled.

T}

$0002	Result overflowed.
$0003	Divide by zero error.
$0004	Invalid op operand passed to FPU. (This error should not ever occurr.)
$0005	FPU passed an operand that is not a valid floating point value.
$0006	Result was inexact with *trap on inexact result* enabled.
$4000	Driver called with invalid operand (>25) in D2.
$8000	FPU failed to complete an operation.

Floating Point Functions

FADD

FP Function 0

Description

Add two single precision numbers.

FSUB

FP Function 1

Description

Subtract two single precision numbers.

FMUL

FP Function 2

Description

Multiply two single precision numbers.

FDIV

FP Function 3

Description

Divide two single precision numbers.

FCMP

FP Function 4

Description

Compare two single precision numbers.

FNEG

FP Function 5

Description

Negate a single precision number.

FABS

FP Function 6

Description

Take absolute value of a single precision number.

FItoF

FP Function 7

Description

Convert integer to single precision floating point.

FFtoIr

FP Function 8

Description

Round single precision floating point to integer.

FTtoIt

FP Function 9

Description

Truncate single precision floating point to integer.

FFtoIt

FP Function 10

Description

Floor function for single precision numbers.

FFtoD

FP Function 11

Description

Convert single precision number to double precision.

FDtoF

FP Function 12

Description

Convert double precision number to single precision.

FDADD

FP Function 13

Description

Add two double precision numbers.

FDSUB

FP Function 14

Description

Subtract two double precision numbers.

FDMUL

FP Function 15

Description

Multiply two double precision numbers.

FDDIV

FP Function 16

Description

Divide two double precision numbers.

FDCMP

FP Function 17

Description

Compare two double precision numbers.

FDNEG

FP Function 18

Description

Negate a double precision number.

FDABS

FP Function 19

Description

Take absolute value of a double precision number.

FItoD

FP Function 20

Description

Convert an integer to double precision floating point.

FDtoIr

FP Function 21

Description

Round double precision floating point to integer.

FDtoIt

FP Function 22

Description

Truncate double precision floating point to integer.

FDtoIf

FP Function 23

Description

Floor function for double precision numbers.

FsetStat

FP function 24

Description

The value in D0 is written into the 32081´s Status Register. Bits 7 and 8 may be used to specify a rounding mode. Bits 9-15 may be used to store an arbitrary value. No other bits have any effect if set. Note that changing the rounding modes will have a global effect on all processes using the floating point processor.

FgetStat

FP Function 25

Description

The value of the 32081´s status register is returned in D0.

Index