68xxx UniFLEX® Programmer's Guide

MANUAL REVISION HISTORY

Revision	Date	Change
A	07/84	Original Release
B	02/86	Manual Update, Updated documentation for the 68xxx microprocessors and corrected miscellaneous errors.
C	09/86	Manual Update for Version 2.0 of 68xxx UniFLEX.

Contents

1.0 Introduction

The UniFLEX Programmer´s Guide describes how to develop programs and utilities that execute under the UniFLEX Operating System. The language used throughout the guide is 68000 assembly language; system programming in higher level languages will be discussed in separate manuals appropriate to those languages. The programmer´s guide is not a complete step-by-step instruction guide for the beginning programmer. A beginning programmer should not attempt to write UniFLEX system programs in assembly language. Rather, this guide is a brief and general introduction to writing simple utilities. It assumes that a knowledgeable systems programmer will be able to build on the information provided here and with experience and the information provided in the Introduction to UniFLEX System Calls be able eventually to develop utilities of most any complexity.

2.0 General

We begin with a general introduction to assembly language programs under UniFLEX: how they run, how they perform system calls, how they handle errors, and what the task environment is like.

2.1 How UniFLEX Programs Run

The most common way a program or utility is run under UniFLEX is by typing the name of such a program in response to a prompt from the UniFLEX shell program. The shell program assumes the name which was typed is a file containing an executable binary program (exceptions such as command text files and precompiled BASIC programs do exist, 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 calling line. When it is finished, the program terminates, returning control to the shell program.

Every program that runs on the system is a task. There may be many tasks active under UniFLEX 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. The 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 CPU's space. When the task terminates, its allocated memory is returned to the system and control passes to the parent task (the task which created or initiated the terminating task).

Our question, then, is how to write the program which the shell program can load and execute, how this program can communicate with the user, system, other tasks, and so forth, and how to terminate the program's execution.

2.2 Introduction to System Calls

When a user's program wishes to communicate with the user, a disk file, another task, or anything else in the system, it does so through calls to the UniFLEX Operating System. The UniFLEX Operating System, also referred to as "the system", is essentially another task which is always available and which has built-in routines to perform a multitude of system-oriented functions. These functions include reading files, writing files, seeking to file locations, setting permissions, creating pipes, reporting task identification numbers (task IDs), creating tasks, terminating tasks, mounting devices, reporting the time, and so forth. A user program can execute these 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 68xxx UniFLEX Assemblers.

2.2.1 The "sys" Instruction

The 68xxx UniFLEX Assemblers have a built-in instruction to make system calls to UniFLEX. It is the "sys" instruction and has the following format:

```
      sys <function_code> [<param_list>]
```

where <function_code> is simply a numeric code for the desired system call. The number and type of parameters required depend on the particular system call. The number of parameters in the list may range from 0 to 4 inclusive. The numeric code is a 16-bit value while parameters are always 32-bit values. Many of the system calls also require certain values or parameters to be in one or more of the 68xxx CPU's registers before executing a "sys" instruction. This depends on the particular system call involved. In those cases where 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 it is necessary for the system call to return one or more values to the calling program. This is generally done by returning the values in selected registers of the CPU. In some cases the returned value or values will be placed at a location specified as one of the input parameters.

The possible system calls under UniFLEX are individually listed and described in the manual called Introduction to UniFLEX System Calls. Along with the description, the necessary parameters and returned values are specified. As an example, look at the "read" system call in that manual. The section on assembly language syntax shows the following:

Expected

```
<file_des> in D0
```

Syntax

```
sys read,<buf_add>,<count>
```

Returns

```
<bytes_read> in D0
```

These statements should be interpreted as follows. Before executing the "read" system call, the programmer must ensure that the appropriate file descriptor (we will find out what a file descriptor is later) is loaded into the 68xxx's D0 register. Next we see the actual "sys" instruction and that in addition to the "read" function code itself, we 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 the "read" call has been executed, the actual number of bytes read will be returned in the 68xxx's D0 register.

All user-accessible 68xxx 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 file "/lib/sysdef" defines the function code for each system call. To see what a particular function code is, you can simply list the file with the command:

```
list /lib/sysdef
```

This file was provided on disk, however, so that the programmer can include those definitions in the source code by using a "lib sysdef" instruction.

It is not necessary to know how the "sys" instruction works, but a brief description might help the programmer's overall understanding. The "sys" instruction generates a "trap #<num>" instruction where the value of <num> is system-dependent but is usually 15. This instruction causes the hardware to switch from user space to system space. The operating system then picks up the handling routine for the specified trap, executes the designated system call (which it obtains from the code directly following the "sys" instruction), and switches back to user space.

2.2.2 Example of a System Call

First of all, let's try a sample program which requires the inclusion of a system call. The simplest program we can write is one which does nothing at all. As soon as it is initiated, it terminates. Thus, the only system call we will need is the call to terminate, "term". Looking in the manual <u>Introduction to UniFLEX System Calls</u> we see that the "sys" instruction itself requires no parameters besides the function code, but that we must put a status value in the D0 register before performing the call. As the description states, if there are no errors this status should be 0. Thus, we can write the following extremely simple UniFLEX program:

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

The "lib sysdef" includes the definitions of all system function codes so that we can specify the "term" call as a symbol ("term") and not have to type in the particular number for that call. The "move.l #0,d0" puts the status in D0 as required by the "term" system call. Then we have the system call itself, which terminates the program. In the case of the "term" system call, control does not return to the calling program after execution. Of course, that is the reason for the system call; 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" instruction includes the symbol "start". This tells the assembler what the beginning location for execution is.

Let's assume you call the source file "nothing.a" and assemble it with the following command:

```
++ rel68k nothing.a +ls +o=nothing.r
++ load68k nothing.r +o=nothing
```

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

```
++ nothing
```

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

2.2.3 Indirect System Calls

We have seen that the "sys" instruction in the assembler is the method by which system calls are made. One quickly notices that in order to use the "sys" instruction directly, all the parameters must be defined at assembly time. Often the parameters cannot be known at assembly time because they will be determined or changed during the execution of the program. Look at the "read" system call we examined earlier. Let's assume we do not know how many characters to read when we are writing the program. That number will be determined by some value the user of our program inputs during its execution. Using the "sys" instruction, our only recourse would be to have the program put the determined number into the appropriate bytes of the program following the software interrupt generated by the "sys" call. This is an unacceptable option because the code would be self-modifying. A solution to this dilemma has been built into UniFLEX in the form of indirect system calls. There are two such forms, and they are themselves system functions which are 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 but not program code. This allows the executing program to put parameters into those locations without creating self-modifying code.

The first of these indirect system calls is called "ind". As described in the Introduction to UniFLEX System Calls, it syntax is as follows:

```
        sys ind,<label>
```

where <label> is the address of the memory locations that will contain the appropriate function code and its parameters. Thus when this system call is executed, the system goes to location "label" and there picks up the desired function code and any necessary parameters. That system call is then executed and when it is complete, control returns to the statement following the "sys ind,<label>" instruction.

To illustrate, let's assume we have a program which needs to read from a file, but does not know how many characters to read until it is executing. We'll assume that somewhere in the first part of the executing program the number of characters to read is determined and stored in a label called "rcount". We will not show the entire program, just those portions required to illustrate the indirect system call.

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

At this point the reader should not be concerned with details of how the "read" call really works or what the file descriptor is. We simply wish to show how the indirect system call is made.

The second form of indirect system call is the "indx" system call. It is similar to the "ind" system call, but instead of providing the label to "ind" which points to the parameters in memory, the "indx" system call assumes that the pointer to the parameters is in the A0 register. To see how this works, the previous sample for the "ind" system call can be repeated 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 obviating 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           Get 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
                ...
                ...
```

One important thing to note in this example is the importance of the order in which the parameters are pushed onto the stack. It is also important to note the "lea 10(a7),a7" instruction following the system call. It removes the parameters which were pushed onto the stack so that the stack is where it was before the section of code containing the system calls.

2.3 System Errors

Upon completion, UniFLEX system calls return to the calling program with an error flag. This flag is the carry bit in the 68xxx condition-code register. If the bit is 0 on return, it implies that no error occurred. If the bit is set (a 1), an error has occurred, and the D0 register contains an error number. The 68xxx UniFLEX Assemblers support two special mnemonics for testing the error status on return from a system call. They are "bes" for "branch if error set" and "bec" for "branch if error cleared". These are equivalent to the standard mnemonics "bcs" and "bcc".

The file "/lib/syserrors" defines the correspondence between the names and numbers of UniFLEX errors. It also contains a brief description of the most general cause of each error. A user can incorporate the labels from this file into a program by including the file in the source code with a "lib syserrors" instruction. Note that UniFLEX does not report errors directly to the user; rather, the system calls return errors in the D0 register. It is the responsibility of the program to report such errors or to handle them as required by the specific application.

2.4 The Task Environment

Under UniFLEX, 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 under UniFLEX, a new task is generated and the program becomes that task. Several tasks may, of course, be "active" in the system at once. By active we mean they have been started, that the system knows about them, and that it is keeping track of them. Only one task or program is actually executing at any given instant. Those tasks which are active but not executing are mapped out of the CPU´s available address space. Whenever that executing task performs some I/O or a system call that will require it to wait, it 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 will force it out so that other tasks can get some execution time. In this manner, the operating system can execute multiple tasks at what seems to be the same time. The switching of tasks occurs so rapidly that the user is generally unaware that the computer is being shared. To assist in keeping track of all the active tasks, UniFLEX assigns a unique task identification number (task ID) to each task. This 15-bit unsigned value is used to identify the task. The system call "gtid" may be used to obtain the task ID if desired.

2.4.1 Address Space

The addresses which can be generated by a 68xxx program make up what is known as the logical address space. Under a UniFLEX system with 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 used by the program to the proper physical addresses.

Under UniFLEX the logical address space is divided into three sections: text, data, and stack. 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 used by the program. It can be both read from and written to during execution. The system stack is located in the stack section. The amount of memory assigned to the task is determined when the program is loaded. The maximum amount that may be assigned is hardware-dependent. Within the confines of the hardware, the user may select with an option to the linking-loader or with the "headset" command any of the following sizes: 128K, 256K, 512K, 1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M, 256M, 512M, 1G, 2G, 4G, S, M, L, small, medium, or large. The size of a task specified by 's' (or "small"), 'm' (or "medium"), or 'l' (or "large") is vendor-dependent. Typically, however, 's' specifies 128K; 'm', the size of physical memory; 'l', the maximum size allowed to a task. The default size is the larger of 128K or the sum of the sizes of the text, data, and bss segments at the time of loading, rounded up to the nearest valid size. Unless otherwise specified by an option to the linking-loader, the data segment starts at the 4-Kbyte page-boundary following the end of the text segment. The data segment grows up in memory during execution as necessary. The stack starts at the top of the memory allocated to the task and grows down in memory during execution as necessary.

2.4.2 Arguments

It is often desirable to pass arguments or parameters to a program when you begin its execution. UniFLEX accomplishes this with the "execl" system call, which is used to begin execution of a program or binary file. At this point we are not concerned with how the arguments are provided to "execl", but rather with how the program which is initiated can obtain those arguments. In other words, if we assume our program has somehow been loaded into memory and execution has started at the beginning of the program, how do we get at the arguments which may have been passed to us?

We find that the system passes arguments to a program by leaving them on the system stack. When the program is initiated, the 68xxx´s supervisor 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 arguments themselves are simply strings of characters which the program must know how to use. In order to find these strings easily, we are also given a list of pointers to the beginning of the strings. In addition,we are given a count of how many arguments have been passed. This 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 0.
2) Just above the argument count (higher addresses in memory) is the list of pointers to the argument strings. These pointers are the 32-bit addresses of the actual strings.
3) At the end of the list of pointers are four null bytes which signify the end of the list (a null pointer).
4) The actual string arguments begin above the null bytes. Each argument string is the string of characters that make up the argument followed by a null byte.

An actual example should help clarify this structure. Let´s assume that whoever started our task passed us three parameters, the name of our program, the name of a file, and an option which starts with a plus sign. It is a UniFLEX convention that the name of the program or command being executed is always passed as the first argument (argument number 0). Assume the program name is "pile", the specified file is "data2", and the option is "+b". Our argument count will be three. Let us arbitrarily say the system stack pointer is at $FFFFFDE0. We should see the following data on the stack:

item	location	contents
arg 2 terminator	$FFFFFE01	$00
argument 2	$FFFFFDFF	'+b'
arg 1 terminator	$FFFFFDFE	$00
argument 1	$FFFFFDF9	'data2'
arg 0 terminator	$FFFFFDF8	$00
argument 0	$FFFFFDF4	'pile'
arg list terminator	$FFFFFDF0	$00000000
pointer to arg 2	$FFFFFDEC	$FFFFFDFF
pointer to arg 1	$FFFFFDE8	$FFFFFDF9
pointer to arg 0	$FFFFFDE4	$FFFFFDF4
argument count	$FFFFFDE0	$00000003

Thus if we wish to get the second argument (argument number 1), we read the pointer stored at the stack pointer + 8. That value is $FFFFFDF9. That is the pointer to the argument string itself and there we find the string of characters "data2" followed by a null byte.

In general, programs or utilities that a system programmer writes will be initiated by the shell program. Specifically they will be started when the user types the name of that program in response to the shell program's prompt. The shell program starts the program by performing an "execl" system call. The arguments that the shell program sets up for the "execl" call (which are those passed to the program) are the arguments that are typed on the shell program's command line after the program name. By convention, the shell program sets argument 0 to be the command or program name itself. The arguments after the program name are then numbered sequentially beginning with one. If our "pile" program above were an executable binary file, the arguments described above would result from a shell command line that looked like this:

```
++ pile data2 +b
```

Thus you can see how UniFLEX utilities obtain input arguments or parameters from the calling line.

One further point might be made regarding the method by which the shell program passes arguments to a command. In particular we are referring to pattern-matching. Three disk files named "file1", "file2", and "filename" would all be listed in response to the command:

```
++ list file*
```

This is due to the pattern-matching feature of the shell program, which allows the asterisk, '*', to match any string of characters except one beginning with a period, '.'. The interesting thing to note is how the "list" command actually knows to list those three files. The answer is that the shell program does not pass "file*" as an argument to list but rather searches the directory for all names that match and passes each one as an individual argument. In response to the command "list file*", the "list" program would see four arguments:

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

Recall that argument number 0 is always the name of the program or command being executed.

3.0 Initiating and Terminating Tasks

Under the multi-tasking environment of UniFLEX, it is possible for one task to 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. This section covers these techniques.

3.1 Terminating a Task

Tasks or programs under UniFLEX are terminated with the "term" system call. When this system call is executed, the task is halted and its memory is relinquished to the system. Before calling the "term" system call, the programmer must place an error status value in the D0 register. When the task terminates, this value is passed to the task's parent. A value of 0 indicates that the task terminated without error. If the task is terminating due to a system error such as an I/O error, the error number returned by that system call should be used as the error status for the "term" system call. If the task is terminating 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 a $000000FF. By convention the parent task would recognize this as a user-defined error. The parent would know some error had occurred causing 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.

3.2 The "wait" System Call

The "wait" system call is issued by a task when it wishes to wait for one of the child tasks it has spawned to terminate. The parent task receives the termination status of its child task through the "wait" system call. The syntax of this call is as follows:

```
sys wait
```

When the system call returns, the termination status is found in the A0 register; the task ID of the terminated task is found in the D0 register.

If there are no child tasks when a "wait" call is issued, the system returns an error. If there is a child task that is still running when the parent issues a "wait" call, the parent will be put to sleep until the child task is finished and has terminated. If a child task finishes and terminates before its parent has issued a "wait" call, the system will save the child task's ID and termination status until the parent does issue a "wait". If several child tasks have been spawned, the parent must issue an individual "wait" call for each one.

The termination status is returned in the lower half of the A0 register and is comprised of two parts, the upper byte and the lower byte. 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 nonzero, an error caused the task to terminate. Under normal conditions, the high-order byte of the termination status (bits 8-15 of A0) is 0. If the high-order byte is nonzero, it indicates that the task was terminated by a system interrupt. In such a case the least significant 7 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 will be described later.

3.3 The "exec" System Call

Generally a user's program will be a binary file on disk. To begin execution of that program, the user simply types its name in response to the shell program's prompt. The shell program then takes charge, loads the program, and begins execution of it. There are times, however, when a user-written program may wish to load and execute a program by itself without going back to the shell program. The tool used to load and execute another program or binary file is the "exec" system call. That is the very function which the shell program uses when it loads and executes a program (remember the shell program itself is just another program). If the "exec" call is successful (that is, no errors occur), the system discards the program which made the "exec" call, loads the new program (a binary file) into memory, and executes it. The task ID of the original task is retained for the new task. Thus, if the "exec" call succeeds, it is impossible to return to the original task. If, however, there is an error in attempting to perform the "exec" system call, the system does not load the new program but rather returns an error number to the calling program, which is still intact. Therefore, a properly written program should follow any "sys exec" call with error-handling code.

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. The syntax is as follows:

```
      sys exec,<file_name>,<arg_list>
```

where <file_name> is a pointer to the name of the file (the name is a string of appropriate characters somewhere in memory which is terminated by a null byte) and <arg_list> is a pointer to a list of argument pointers. In other words, <arg_list> is an address at which is stored a list of pointers. This list of pointers is composed of consecutive 4-byte addresses, or pointers, to the actual argument strings. The list is terminated by four null bytes, which could be considered a pointer to 0. Each pointer in the list is the address of the actual argument string which is terminated by a null byte. When the "exec" system call is complete, the new program will have these arguments available in the exact format described in Section 2.4.2.

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

```
                 text
                 lib    sysdef

      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 occurs.
      * Note the D0 register already has the error from "exec".

                 sys         term

      * strings and data

                 data
      filen      fcc         ´/bin/ls´,0
      arg0       fcc         ´ls´,0
      args       fcc         ´+ba´,0
      args       ds.l        arg0,arg1,0

                 end         start
```

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

3.4 The "fork" System Call

The "fork" system call provides the only way of spawning a new task under UniFLEX. When a program invokes the "fork" system call, the system creates a new task, which is almost identical to the old one (the old task still exists). This new task has the same memory and stack allocation, the same code in the memory space, the same open files, pointers, and so forth. Immediately after forking you essentially have two identical tasks or programs running on the system. Now, usually you want the new task to do something different. Consequently, in most cases the new task will immediately perform an "exec" call to load some program from disk and execute it. This is the technique used by the shell program to start a background job. When the shell program sees a command ending with an ampersand ("&"), instead of directly doing an "exec" it does a fork to create a second shell program. Now the newly created shell program does an "exec" of the desired command while the old shell program is still around to accept further commands.

The syntax of the "fork" system call is as follows:

```
      sys fork
```

The "fork" call requires no parameters. The tricky 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 will resume in each of the two programs. The difference is in where that execution resumes. In the new task, execution resumes at the instruction immediately following the "fork" 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 by means of a fork, 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 (requiring only 2 bytes). The reader should also 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 will not be created and the system returns an error number to the old task (still 2 bytes past the "fork" system call).

A section of code (not an entire program) will help 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 for error, branch if so
           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 old task waits for the new one (its child) to finish before continuing. That is the purpose of the "wait" system call at "prwait". Note that the "wait" system call returns the terminated task's ID in the 68xxx D0 register.

4.0 File Handling

The manipulation of files, terminals, directories, printers, and any other device is perhaps the most important part of the assembly language interface to UniFLEX. It is, therefore, imperative for the system programmer to have a good understanding of the material in this section.

4.1 General UniFLEX File Definitions

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

4.1.1 Device-independent I/O

Under UniFLEX, anything outside the program's memory to which the program can write or from which it can read is treated identically. A file on disk is treated in the same way that a terminal is treated. A terminal is treated exactly like a pipe or a printer spooler. This feature, which is termed device-independent I/O, allows a program that sends its output to a terminal to send its output to a disk file, printer spooler, pipe, or any other device on the system. Device-independent I/O lends a great amount of versatility to the system and simplifies program development and maintenance.

Device-independence is made possible by the device driver routines. Each of these routines creates a standard interface between the system and the device for which it is written. One routine opens the device; one closes it. These routines permit the system to do anything necessary to the device to prepare it for reading and writing and to finalize anything necessary when all I/O is complete. The two most important device driver routines are the read and write routines. They permit the caller to read data from or to write data to the device.

4.1.2 File Descriptors

A user who wishes to perform some operation on a UniFLEX file informs the system which file to operate on by providing a "file descriptor". (We use the term "file", but because of device-independence it can refer to a disk file, terminal, pipe, or any other device). The UniFLEX file descriptor is a 4-byte numeric representation of a specific file or device. The system assigns the file descriptor when it opens or creates the file. UniFLEX then keeps track of which file descriptors correspond to which files. In this way, the user need only supply a number instead of the name of the file each time the file is to be referenced.

As an example, look back at the description of the "read" system call. You will see that this system call requires the program to place a file descriptor 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 descriptors range from 0 to 31 inclusive. No task may have more than thirty-two files open at a time.

4.1.3 Standard I/O Files

When the shell program begins execution of a task, it automatically assigns three I/O files to that task: standard input, standard output, and standard error. Standard input is the file from which a command takes its input. Standard output is the file t which a command sends its output. Standard error is the file to which many error messages are directed. The system assigns a file descriptor of 0 to the standard input file and opens the file for reading. It assigns a file descriptor of 1 to the standard output file and opens that file for writing. It assigns a file descriptor of 2 to the standard error file and opens that file, too, for writing. By default, the system uses the user´s keyboard as standard input and the user´s display for both standard output and standard error.

Because the system opens these standard I/O files and assigns them each a file descriptor, the program does not have to perform any "open" or "create" calls in order to use them. As soon as a task begins running, it can perform a "read" system call with a file descriptor of 0 (standard input) or a "write" system call with a file descriptor of 1 or 2 (standard output or error output).

One nice thing about the standard I/O files is that they can be "redirected" without any change to the program whatsoever. The symbol ´<´ tells the shell program to redirect standard input to the file whose name follows the symbol. Similarly, the symbols ´>´ and ´%´ redirect standard output and standard error. The file to which standard input is redirected must already exist. However, if the file to which standard output or standard error is redirected does not exist, the system creates it. In fact, if the file does already exist, the system deletes the contents of the file before executing the command. To avoid this effect, the user may instead direct the shell program to append data to the file specified as standard error or standard output by duplicating the symbol used for redirection: ">>" or "%%".

The program need not be concerned with the nature of the devices that are designated as the standard I/O files. They may be the user´s terminal, a disk file, a pipe, or something else. Because of device-independence and the fact that the program knows that the file or device (whatever it may be) has previously been opened, the program simply performs the I/O operations without caring what the I/O files actually are.

4.2 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 I/O manipulations with a file, it should generally close that file. A program also needs the ability to create new files on the system. This section addresses those operations in some detail.

4.2.1 The "open" System Call

In order to read or write to an existing file or device, we must first open that file, no matter what the device is. The syntax of the "open" system call is:

```
      sys open,<file_name>,<mode>
```

where <file_name> is a pointer to a zero-terminated string containing the name of the file to be opened, and <mode> is a number which determines whether to open the file for reading, writing, or both. The value of <mode> may be 0, 1, or 2. If <mode> is 0, the file is opened for reading only; if 1, for writing only; if 2, for both reading and writing. On return from the "open" call, the 68xxx D0 register will contain the 4-byte file descriptor assigned to that file. All future references to the file will be made by means of this file descriptor. An error will be returned from the "open" 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 open, or if the path leading to the file cannot be searched.

4.2.2 The "close" System Call

When a task terminates, UniFLEX 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. Because the system can have only a finite number of files at one time, closing a file makes room for the system to open another file. Furthermore, in the event of a system crash, you will be better off having closed any files which no longer required I/O. The "close" system call is performed by loading the file descriptor of the file you wish to close in the D0 register, then performing a "sys close".

4.2.3 The "create" System Call

The "create" system call is used to create disk files only. To create directories, pipes, devices, and so forth, other system calls must be used.

When the system creates a file, it sets a permission byte which determines what type of access users will have to the file. The permissions for the user who owns the file are independent of the permissions for all other users on the system. The system manager always has complete access to all files.

Every task has associated with it a default permissions byte, which may be altered wi th either the "dperm" command or the "defacc" system call. Every new file or task that is created by that task is subject to the default permissions at the time the "create" system call is invoked. The permission mask specified in the "create" call allows you to deny permissions which the default permissions grant, but does not allow you to grant permissions that the default permissions deny.

The syntax of the "create" system call is as follows:

```
      sys create,<file_name>,<perm_mask>
```

Once again, <file_name> is a pointer to a zero-terminated string containing the name of the file to create. The file will be created in the working directory unless the user explicitly specifies another directory. The argument <perm_mask> is a symbolic name or a numeric value which permits the user to modify the default permissions on the new file. If the file does not already exist, the system performs a logical "and" of the default permissions byte and this permissions mask to determine the actual permissions for the file. The reader should refer to the Introduction to UniFLEX System Calls for more details of setting these permissions.

If the file already exists, the system truncates it to zero length by deleting all existing data. In such a case, the file retains the original permissions regardless of the <perm_mask> supplied to the "create" call. In other words, if the file <file_name> already exists, the system ignores the permission mask.

4.3 Reading and Writing

Perhaps the most commonly used system calls are "read" and "write". A program uses these system calls to communicate with the user, disk files, printers, other tasks, and anything else in the outside world. It is imperative for the would-be UniFLEX system programmer to have a good understanding of these calls, their use, their requirements, and their characteristics. Reading and writing under UniFLEX are elementary procedures, and, as such, they permit great versatility in the way in which files are accessed. When speaking of a disk file, a 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. Thus, both sequential and random access of files are quite simple.

The "read" and "write" system calls assume that 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. We will see how it can be set in the section on seeking. The only parameters required, then, to read from or write to a file are the file descriptor specifying the particular file, the count of characters to be read or written, and a the address of a buffer in memory to read into or write from. We shall look at each call separately.

4.3.1 The "read" System Call

To execute a "read" system call, the programmer must load the D0 register with the file descriptor number before making the call with the following syntax:

```
      sys read,<buf_add>,<count>
```

where <buf_add> specifies an address in the user program's memory where the data read from the file should be placed and <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. On return from the "read" system call, the number of bytes which was actually read is in the D0 register.

When dealing with a regular disk file, the system will always read <count> bytes if possible. In two cases it cannot do so. If the program attempts to read past the end of the file, the system returns the number of characters read. For example, if we have a file of only 120 characters, and a "read" call is issued with a <count> of 256, the "read" call will succeed but will show that only 120 characters were actually read. After this call the file position pointer will be left pointing to the end of the file. Any subsequent "read" call will return with no error, but the number of bytes read will be equal to 0. In fact, a program should detect an end-of-file condition by the successful completion of a "read" system call with the number of characters read being 0.

If a physical I/O error occurs, rather than returning the number of bytes read, the system returns -1 to indicate the error.

Reading and writing to terminals is handled in the same manner as reading and writing disk files. Identical system calls are used for terminals and disk files. However, if the file being read is a terminal, the system returns at most one line. A line is all the characters up to and including the carriage return typed since the last carriage return. Thus, even if the system executes a "read" system call with a <count> of 1024, if the user is at a terminal and types the letters "halt" followed by a carriage return, the "read" call would return with 5 as the number of bytes read. If the user has not typed anything when the call is issued, the calling program will be required to wait until something is typed. As with regular disk files, it is possible to detect an end-of-file condition from a terminal by performing a "read" call and receiving no error and a count of 0. An end-of-file condition from a terminal is produced by typing a control-D as the first character in a line. Note that the control-D itself is not passed to UniFLEX, 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 1,024 bytes of data, placing them in a buffer named "buffer". We assume the file has already been opened for read and that 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
          ...
          ...
```

On return from the "read" system call, we first check to see if it returned an error. If it did, we assume that the program handles it properly at "rderr". If the call does not return an error, we check for an end-of-file condition. Recall that an end-of-file condition is recognized by a returned value of 0 from a successful "read" system cal. If we are at the end of the file, the program jumps to "endof" where, once again, we 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 1 byte 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.

4.3.2 The "write" System Call

The "write" system call is executed by first loading the D0 register with the appropriate file descriptor, then issuing an instruction of the following form:

```
      sys write,<buf_add>,count
```

where <buf_add> specifies an address in the user program´s memory where the data to write to the file are located and <count> is the number of characters to write to the file. On return from a successful "write" system call, the number of bytes written is in the D0 register. It is not necessary to compare this value with <count> because if the call was successful, the values will be the same.

Let´s look at a complete program to send the message "Hello there!" to the standard output file. If an error occurs while writing to that file, we will send the message "Error writing standard output." to the standard error file. Recall that the system assigns a file descriptor of 1 to standard output and a file descriptor of 2 to standard error.

```
          text
          lib       sysdef       include system definitions

* start of main program

sayhi     move.l    #1,d0        write to std. output
          sys       write,hello,hlng  send message
          bec.s     done         exit if no error
          move.l    d0,-(a7)     else, save error number
          move.l    #2,d0        write to std. error output
          sys       write,erm,elng  sen error message
          move.l    (a7)+,d0     restore error number
          bra.s     done2
done      move.l    #0,d0
done2     sys       term         terminate program

* strings

          data
hello     fcc       ´Hello there!´,$d,0
hlng      equ       *-hello      compute length of string
erm       fcc       ´Error writing standard output.´,$d,0
elng      equ       *-erm        compute length of string

          end       sayhi        give starting address
```

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

4.3.3 Efficiency in Reading and Writing

A system programmer can do several things to achieve efficient reading and writing of files under UniFLEX. The first and most obvious is to read or write as much of a disk file as possible with a single call. Much less system overhead is involved in executing one call to read 4,096 characters than in executing thirty-two calls to read 128 characters each. The most efficient "read" and "write" calls are those made in multiples of 512 bytes. This is, of course, because the size of a disk block under UniFLEX is 512 bytes. Due to the way the system implements memory mapping, a programmer can achieve additional efficiency by placing each buffer on a 512-byte boundary.

By all means do not perform single-character I/O with a system call for each character. If you need single-character, the program should handle the necessary buffering such that system calls are made only when a buffer is full.

4.4 Seeking

UniFLEX maintains a pointer, which indicates the current position for reading or writing, for each open disk file. The program can manipulate this pointer to point to any character in the file by using the "seek" system call. The "seek" call is really only useful on disk files. Before making a system call to seek, the user must load the appropriate file descriptor in the D0 register. The syntax of the "seek" call is as follows:

```
      sys seek,<position>,<mode>
```

where <position> is a four-byte signed offset from the point of reference in the file determined by <mode>. A positive number indicates seeking toward the end of the file; a negative, toward the beginning. The value of <mode> may be 0, 1, or 2, as shown in the following table:

Value of <mode>	Meaning
0	Beginning of the file
1	Current position in file
2	End of the file

On return from the "seek" call, the new current position relative to the beginning of the file is in the D0 register. To find the current position in a file, you can look at the value returned by the system call "sys seek,0,1".

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 32,000 records in our file, so the record number can be represented in 16 bits. We wish to write a subroutine which will read the record specified by the record number in the D0 register 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. We then seek to that position and read 256 bytes. Our routine looks like this:

```
                ...
                ...
      getrec    move.l    a0,iread_cnt    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_cnt    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
      iseek_cnt dc.l      0               seek address (unknown)
                dc.l      0               type 0: position from begin
      iread     dc.w      read            read function code
      iread_cnt 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 know neither to 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 areas of memory (at "iseek" and "iread") where these values can be stored when the program executes and determines just what they are.

4.5 File Status Information

Certain information about each file or device is available to the user through the "status" and "ofstat" system calls. The two calls differ in that "ofstat" is used to obtain information about an opened file whereas "status" obtains information about an unopened file. The syntax for "ofstat" is as follows:

```
<file_des> in D0
sys ofstat,<buf_add>
```

where <buf_add> specifies an address in the user program´s memory where the system should place the data returned by "ofstat". This buffer must be at least 22 bytes long. The information returned in the buffer is the same as that returned by the "status" system call except tha "st_cnt" is not the link count but the number of tasks connected to a pipe. The user must load the D0 register with the appropriate file descriptor before invoking the "ofstat" system call.

The syntax for "status" is as follows:

```
sys status,<file_name>,<buf_add>
```

where <file_name> is a pointer to a zero-terminated string containing the name of the appropriate file and <buf_add> specifies an address in the user program´s memory where the system should place the data returned by "status". This buffer must be at least 22 bytes long.

When the "status" system call is completed, the buffer will contain all the information available about the file. The file "/lib/sysstat" defines this buffer as follows:

```
 base 0 Set initial values

st_dev  ds.w  1  Device number
st_fdn  ds.w  1  Fdn number
        ds.b  1  Filler
st_mod  ds.b  1  File mode
st_prm  ds.b  1  Permission bits
st_cnt  ds.b  1  File link count
st_own  ds.w  1  File owner´s user ID
st_siz  ds.l  1  File size in bytes
st_mtm  ds.l  1  Time of file´s last modification
st_spr  ds.b  4  Spare--for future use only

ST_SIZ  ds.w  0  Size of status buffer
```

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. Every file on the system has a file descriptor node, which is a block containing information about the file, including its location. It is from the fdn that "status" and "ofstat" obtain their information. The file and permissions will be explained in the next paragraph. The link count is the number of directory entries that are linked to the fdn. More information on linking can be found in Section 5. The file owner's user ID is a 2-byte ID that was assigned to the user by the system manager with the user's user name. The file size in bytes is the exact number of characters in the file. The time of last modification is the internal UniFLEX representation of the last time someone wrote to the file.

The bytes describing both file type and permissions are flags in which the state of the individual bits is used to convey information about the file. The byte describing the file type looks like this:

file mode (st_mod):

```
--------------------------------
! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
--------------------------------
      !   !   !   !   !   !   !
      !   !   !   !   !   !   ----- regular file
      !   !   !   !   !   --------- block device
      !   !   !   !   ------------- character device
      !   !   !   ----------------- directory
      !   !   --------------------- spare
      !   ------------------------- spare
      ----------------------------- pipe
```

Notice that only five bits are used in this byte. The low-order bit is always set. If it is the only bit set, the device is a regular file. A block device is a device, such as a floppy disk drive, which handles data in 512-byte blocks. A character device is one which handles data a single character at a time. A terminal is an example of a character device. If bits 0, 1, and 2 are set, the device is a pseudoterminal.

The permissions byte shows what permissions are granted or denied for the file. Its format is as follows:

permissions (st_prm):

```
--------------------------------
! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
--------------------------------
      !   !   !   !   !   !   !
      !   !   !   !   !   !   ----- owner read permission
      !   !   !   !   !   --------- owner write permission
      !   !   !   !   ------------- owner execute permission
      !   !   !   ----------------- others read permission
      !   !   --------------------- others write permission
      !   ------------------------- others execute permission
      ----------------------------- user ID bit for execute
```

In this byte any or all of the seven bits used may be set at one time. If a bit is set, it shows that the corresponding 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 of the file while that file is executing. As an example of the usefulness of this feature, consider a user, "joe", who has a data-base program which manipulates a large data file. Now, "joe" does not want anybody on the system to be able to directly read from or write to his data file, so he denies read and write permissions to others on that file. He does, of course, grant read and write permissions for himself (the owner). Even though he does not want anyone to be able to read and write his data file directly, "joe" would like other users to be able to run his data-base program, which manipulates the data file. All he needs do is set the "user ID" permission bit in his data-base program. With the "user ID" bit set, any users who run the data-base program have the same permissions as "joe". Thus, they can manipulate the data file while running the data-base program. As soon as the data-base program is terminated, however, the other user no longer has 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, which is available to all users under UniFLEX. A directory is a special type of file, and the only way to create one is by using the "crtsd" system call, which only the system manager may do. Without the "user ID" bit set, 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. Because the owner of "crdir" is the system manager, any user may create a directory.

5.0 Directories and Linking

A UniFLEX directory entry is nothing more than the name of the file and a pointer to the file descriptor node (fdn) for the file. The fdn is a small unit on the disk which contains certain types of 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, for more than one directory entry to point to the same fdn; each of these entries is called a link. If you do a long directory listing on a directory (ls +l), you will find one field in each entry, the link count, which is the number of directory entries which point to, or are linked to, that file. The link count should be one at a minimum; if it ever goes to 0, the operating system deletes the file.

An example of linking can be seen in every directory on a UniFLEX disk. Recall each directory contains two entries called "." and ".." (they do not appear in an "ls" listing unless you use the ´a´ option). The symbol "." represents the directory itself; the symbol ".." represents its parent. Thus, typing "." as a directory name is equivalent to typing the file specification of the working directory. Typing ".." is equivalent to typing the file specification of the parent of the working directory. These entries in the directory are not separate files, but rather are links to the working directory and its parent. That is why every directory on the system has a link count greater than 1.

The "link" and "unlink" system calls allow the programmer to create and break links. The "link" system call is quite straightforward: the user specifies a pointer to the name of the file to be linked to and a pointer to the new name to put into the directory. The system call then creates the appropriate link.

The "unlink" system call is a bit more complicated. The programmer merely provides a pointer to the name of the file to unlink. "Unlink" first removes the specified name from the directory and decrements the link count by 1. Next, it tests to see whether or not the link count is 0. If the link count is 0 and the file is closed, the operating system deletes the file. Otherwise, the operating system does not delete it.

If a file is open at the time an "unlink" call is made, the unlink operation will take place, but the operating system does not delete the file. The user can still read or write to the file as long as it is left open. When the user closes the file, the "close" system call checks the link count. If the link count is 0 and no other user has the file open, the operating system deletes the file. This behavior creates interesting possibilities for a program. A program can open a file and immediately unlink it. 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, the operating system will immediately delete it.

6.0 Other System Functions

This section is devoted to several specialized features and functions which are available to the system programmer. Specific syntax statements will not always be given. It is assumed that the reader can obtain this information from the Introduction to UniFLEX System Calls.

6.1 The "break" and "stack" System Calls

Earlier we learned that when a task is started, it is allocated text, data, and stack memory according to the size of the program. A running task may change the amount of memory allocated to its data or stack space. It is also possible to relinquish allocated memory to the system--that is, to deallocate data or stack memory. Stack allocation and deallocation are performed by the "break" and "stack" commands. When the user supplies an address to the "break" system call, the system attempts to allocate memory so that there is RAM up through the specified address. Because memory is allocated in sections, some memory may exist beyond the specified address. If an address is specified which falls below the amount of program memory already allocated, the surplus memory is relinquished to the system. The "stack" command works in much the same way, except that the stack grows downward in the CPU´s address space.

6.2 The "ttyset" and "ttyget" System Calls

The user may alter and examine several configuration parameters of terminals under UniFLEX. These parameters include such things as the line-cancel character, the backspace character, the length of the delay after carriage returns, mapping of upper- to lowercase, and tab expansion. The configuration of all these parameters is represented in 6 bytes of data. These 6 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 6-byte buffer must be established in memory which contains the desired configurations for "ttyset" or which will receive the information about the current configuration from "ttyget". The file "/lib/systty" defines this buffer as follows:

```
 base 0

tt_flg  ds.b  1  Flags
tt_dly  ds.b  1  Delays
tt_cnc  ds.b  1  Line-cancel character (default is control-X)
tt_bks  ds.b  1  Backspace character (default is control-H)
tt_spd  ds.b  1  Terminal speed
tt_spr  ds.b  1  Stop output byte

TT_SIZ  ds.w  0  Size of buffer
```

The terminal speed byte presently implements only 4 bits. Bits 2, 3, and 4 define the configuration of the terminal; bit 7 is a flag which, when set, indicates that the terminal has input characters waiting to be consumed by the program. This bit is only meaningful when read--that is, the input ready condition cannot be set with this bit and "ttyset". The entire byte looks like this:

```
  terminal speed byte (tt_spd):

---------------------------------
! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
---------------------------------
  !   !   !   !   !   !   !   !
  !   !   !   !   !   !   !   ----- spare
  !   !   !   !   !   !   --------- spare
  !   !   !   !   !   ------------- first bit of terminal configuration
  !   !   !   !   ----------------- second bit of terminal configuration
  !   !   !   --------------------- third bit of terminal configuration
  !   !   ------------------------- spare
  !   ----------------------------- spare
  --------------------------------- input ready to be consumed
```

Under normal input operations the "input ready to be consumed" bit does not come on until an entire line has been input and terminated by a carriage return. Two special input modes can, however, be established in which the "input ready to be consumed" bit will come on as soon as a single character is input. These modes, known as "raw I/O mode" and "single character input mode", are described later in this section.

The following table shows the configuration of the terminal for all possible settings of the terminal configuration bits:

Terminal Configuration (Bit Pattern)	Data Bits	Stop Bits	Parity
0 0 0	7	2	Even
0 0 1	7	2	Odd
0 1 0	7	1	Even
0 1 1	7	1	Odd
1 0 0	8	2	None
1 0 1	8	1	None
1 1 0	8	1	Even
1 1 1	8	1	Odd

The stop output byte contains bits which control the stopping and starting of output to terminals. A user can use one of two methods to stop and start output to a terminal: the escape key and XON/XOFF processing. The method which uses the escape key permits a user to type an escape character (hexadecimal 1B) to stop output. A subsequent escape character restarts the output. The XON/XOFF method permits a user to type an XOFF character (hexadecimal 13) to stop output and a subsequent XON character (hexadecimal 11) to restart it. Many terminals produce XON and XOFF characters automatically to prevent the computer from sending too many characters to the terminal at once. The escape and XON/XOFF mechanisms can be independently enabled or disabled by setting or clearing the proper bits in the byte "tt_spr". The byte looks like this:

```
    stop output byte (tt_spr):

   ---------------------------------
   ! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
   ---------------------------------
     !   !   !   !   !   !   !   !
     !   !   !   !   !   !   !   ----- first bit of baud rate
     !   !   !   !   !   !   --------- second bit of baud rate
     !   !   !   !   !   ------------- third bit of baud rate
     !   !   !   !   ----------------- fourth bit of baud rate
     !   !   !   --------------------- spare
     !   !   ------------------------- any character restarts output
     !   ----------------------------- enable XON/XOFF for I/O
     --------------------------------- disable ESC for stopping output
```

The following table shows the baud rate defined by all possible settings of the first 4 bits of the stop output byte:

Bit Pattern	Baud Rate	Bit Pattern	Baud Rate
0 0 0 0	--	1 0 0 0	1200
0 0 0 1	75	1 0 0 1	1800
0 0 1 0	110	1 0 1 0	2400
0 0 1 1	134.5	1 0 1 1	3600
0 1 0 0	150	1 1 0 0	4800
0 1 0 1	200	1 1 0 1	7200
0 1 1 0	300	1 1 1 0	9600
0 1 1 1	600	1 1 1 1	19200

When set, the bit labeled "any character restarts output" instructs the terminal drivers to restart the output if it has been stopped by either an escape or XOFF.

The delay byte tells the system how long to delay after outputting certain characters. A delay is useful in cases where a slow output device such as a teleprinter, which requires a delay for carriage returns, is attached to the system. Two of the delays can take on any of four different values as specified by a 2-bit value. The other two delays are represented by one bit each and are, therefore, either on or off. The format of the delay byte is as follows:

```
            delay byte (tt_dly):

   ---------------------------------
   ! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
   ---------------------------------
     !   !   !   !   !   !   !   !
     !   !   !   !   !   !   !   ----- new-line delay
     !   !   !   !   !   !   ---------  "    "    "
     !   !   !   !   !   ------------- carriage return delay
     !   !   !   !   -----------------    "      "      "
     !   !   !   --------------------- horizontal-tab delay
     !   !   ------------------------- form feed/vertical-tab delay
     !   ----------------------------- spare
     --------------------------------- spare
```

The new-line delay is performed after each line-feed character (hexadecimal 0A) is output. The length of the delay is determined by the combination of bits set, as shown in the following table.

Bit 1	Bit 0	Length of Delay in Milliseconds
0	0	0
0	1	10
1	0	20
1	1	30

The carriage return delay is performed after each "carriage return" character (hexadecimal 0D) is output. The length of the delay is determined by the combination of bits set, as shown in the following table.

Bit 3	Bit 2	Length of Delay in Milliseconds
0	0	0
0	1	10
1	0	20
1	1	30

The horizontal-tab delay may be either on or off. If on (bit 4 set), the delay is 20 milliseconds long. The form-feed, or vertical-tab delay, may also be either on or off. If on (bit 5 set), the delay is 240 milliseconds long.

The 8 bits of the flag byte represent eight different modes of operation for the terminal. When a bit is set, the corresponding mode is in effect. The format of the flag byte is as follows:

```
            flag byte (tt_flg):

   ---------------------------------
   ! 7 ! 6 ! 5 ! 4 ! 3 ! 2 ! 1 ! 0 !
   ---------------------------------
     !   !   !   !   !   !   !   !
     !   !   !   !   !   !   !   ----- raw mode
     !   !   !   !   !   !   --------- echo input
     !   !   !   !   !   ------------- expand tabs on output
     !   !   !   !   ----------------- map upper- to lowercase
     !   !   !   --------------------- automatic line-feed
     !   !   ------------------------- echo backspace echo character
     !   ----------------------------- single character input mode
     --------------------------------- ignore control characters
```

We shall describe each of these modes separately in the following paragraphs.

Bit 0: Raw Mode

By default, the terminal drivers process various characters on input and output before passing on. However, when the terminal drivers are in raw mode, they do no special processing of the input or output characters. Each and every character typed on the terminal--including backspace characters, line-cancel characters, tab characters, control-C and other control characters--is directly input to UniFLEX. Similarly, every character output to the terminal is output directly: no tab expansion is performed, no line-feed characters are appended to carriage returns, and so forth. In addition, the parity bit is not stripped on either input or output. A program executing in raw mode has complete control of every character input or output and must perform any special processing itself.

In raw mode a "read" system call will not have to wait for an entire line to be input before it can read characters. If a single character is available, the "read" call will return with just that character. It is still possible to read more than one character with a single "read" call but only if the characters have already been typed into the input buffer before the call is made.

Bit 1: Echo Input

By default, the terminal drivers echo each character on the display device as it is input. When the drivers are in echo-input mode, the terminal should be operating in full-duplex. At times a user may wish to disable the echo. For example, when a user logs in, the "login" program writes the message "Password:" on the display, then turns the echo-input bit off while the user enters the password, so that the password is not echoed to the screen.

Bit 2: Expand Tabs on Output

By default, the terminal drivers expand tabs on output. If a terminal's hardware cannot expand tab characters, setting expand-tabs mode allows the terminal driver to do so. The system assumes that tab stops are at 8-column intervals. Thus, if this bit is on, each time a horizontal tab character (hexadecimal 09) is output, UniFLEX will space over to the next column which is a multiple of 8 (unless it is already at such a column).

Bit 3: Map Upper- to Lowercase

By default, the operating system assumes that a terminal has both upper- and lowercase capability and that the user will type most commands and input in lowercase characters. It is possible, however, to use a terminal which supports only uppercase by instructing the terminal drivers to map all input characters from upper- to lowercase and all output characters from lower- to uppercase. To do so, the user simply turns on the "map upper- to lowercase" bit in the "ttyset" flag byte. The operating system automatically turns this bit on if the user name typed in response to a "login" prompt begins with an uppercase letter. Thus, a terminal which supports only uppercase can be connected to UniFLEX without special considerations.

Bit 4: Automatic Line-feed

By default, the terminal drivers will automatically output a line-feed character (hexadecimal 0A) after each carriage return.

Bit 5: Echo Backspace Echo Character

By default, if the backspace character is defined as control-H (hexadecimal 08), the terminal drivers will echo the control-H, followed by a space character and another control-H. This sequence of characters will erase the incorrect character on terminals which do not do so automatically.

6) Single Character Input Mode

By default, UniFLEX process I/O one line at a time. Under such circumstances, 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. However, for certain applications a program might prefer to input one character at a time without having to wait for a carriage return. That behavior is accomplished by putting the terminal drivers in "single character input mode". When this mode is in effect, the program can read a character as soon as it has been typed without having to wait for an entire line and carriage return. It is possible to read multiple characters while in single character input mode, if they are available. When the terminal drivers are in single character input mode, they strip the parity bit off all input characters, but only control-C, control-D, and control-\ are treated as special characters. In other words, tabs, backspaces, and line-cancel characters are ignored. Any processing of these characters must be handled by the program.

Bit 7: Ignore Control Characters

By default, the terminal drivers do not ignore control characters. It is possible, however, to put the drivers in "ignore control characters" mode. When this mode is in effect, the drivers will ignore all control characters which do not have special meaning--that is, all control characters except the following ones:

Carriage Return
Horizontal Tab
control-C
control-D
control-\
Backspace character (if defined as a control character)
Line-cancel character (if defined as a control character)

Those control characters which are ignored will still be echoed if "echo input characters" mode is also in effect.

6.3 Pipes

UniFLEX provides a mechanism called the pipe which 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. If a pair of tasks needs two-way communication, the user must establish two pipes--one to send information 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 descriptors necessary for these write and read operations are provided by the system when it creates the pipe. A pipe is created with the "crpipe" system call.

The pipe mechanism works similarly to a holding tank with valves on the input and output lines. If the tank is not full, the writing task can pump data into it even if 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 if the writing task has the input valve closed (is not currently writing). If the tank is full, the writing task must wait until the reading task has emptied it before it can pump in more data. If the tank is empty, the reading task must obviously wait until the writing task has pumped in some data. Under UniFLEX the holding tank is a 4-Kbyte buffer located on the disk which has been configured as the pipe device. The "tune" command can be used to designate a different pipe device. Each pipe has a buffer, but none of these buffers shows up in any directory.

A section of code will provide a sample of how to establish a pipe between a task ´A´ and its child task, task ´B´. Firstly, the pipe is created with the "crpipe" system call in task A. Next, we do a "fork" system call to create task B. We then set up the file descriptors so that we will be writing from task A to task B. The code would look something like this:

```
...
...
text
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
        ...
        ...
```

The major point to learn from this example is how each task closes the portion of the pipe that it cannot use. As previously stated, a pipe only allows data to be transmitted in one direction. The "crpipe" system call creates a pipe with two file descriptors--one for reading and one for writing. After the system performs the "fork" system call, both tasks have a pipe file open for reading and writing. We assume that the writing task will eventually close the write file descriptor and that the reading task will eventually close the read file descriptor, but we must specifically ensure that the writing task closes the read file descriptor and that the reading task closes the write file descriptor. In fact, these closing operations should be performed as soon as possible, before any reads or writes to the pipe are attempted.

6.4 Program Interrupts

UniFLEX supports a number of "program interrupts", which allow one program or task to interrupt another. This feature permits timing and synchronization among the tasks in the system. It also gives the programmer the ability to terminate tasks prematurely by means of software.

6.4.1 Sending and Catching Program Interrupts

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

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

If the effective user ID of the task executing this code matches that of task number 327 or if the task is owned by the system manager, the system will send a "quit" interrupt to task 327. We will define the quit interrupt and other interrupts in a moment. Notice that the system call used to send program interrupts is "spint". This same system call can be used to send an interrupt to all tasks associated with the terminal executing the program. Consult the documentation of "spint" in the Introduction to UniFLEX System Calls for details.

Often it is possible for a task to "catch", or intercept, a program interrupt when it receives one. 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. The system call that provides the capability of catching a program interrupt is called "cpint". In effect, this system call permits the user to establish an interrupt-vector address so that if a program interrupt is received, control passes to that address. The programmer may place a routine at the address which handles the interrupt in some special way. Certain addresses are specially treated. If the address specified for the caught interrupt is $000000, the default action of the interrupt will be allowed to occur much as if the interrupt had not been caught at all. If the address specified is $000001 or any other odd address, the interrupt will be ignored much as if it had never been sent. Note that no code is actually placed at these addresses; the "cpint" system call recognizes them as special values and performs the indicated interrupt handling without ever jumping to or using them as real addresses. Any even address address other than $000000 is assumed to be a valid address in the program's memory, and control passes to that location. There the programmer places the desired interrupt-handling routine, which must be exited with an "rtr" instruction. When the system executes this "rtr" instruction, control passes to the point in the program where the interrupt occurred.

Once a program interrupt has been caught and processed, the system resets itself to the default condition where interrupts are no longer intercepted. Therefore, to continue catching program interrupts it is necessary to reissue the "cpint" call after each interrupt is processed. Following is a list of the types of program interrupts possible under UniFLEX.

Name	Number	Description	A	C	D	I	R
SIGHUP	1	Hangup	+	+	-	+	+
SIGINT	2	Keyboard	+	+	-	+	+
SIGQUIT	3	Quit	+	+	+	+	+
SIGEMT	4	A-line (Axxx) emulation trap	+	+	+	+	+
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	F-line (Fxxx) emulation trap	+	+	+	+	+
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	Division by 0	+	+	+	+	+
SIGPRIV	24	Privileged instruction	+	+	+	-	+
SIGADDR	25	Address error	+	+	+	-	+
SIGDEAD	26	A child task terminated	-	+	-	+	+
SIGWRIT	27	Write to read-only memory	+	+	+	-	+
SIGEXEC	28	Data or stack space violation	+	+	+	-	+
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 signal	+	+	-	+	+
SIGINPUT	35	Input is ready	+	+	-	+	+
SIGDUMP	36	Take memory dump	0	+	+	+	+
	37-41	System-defined interrupts					
SIGUNORDERED	42	MC68881 branch or set on unordered operand	+	+	-	+	+
SIGINEXACT	43	MC68881 inexact result	+	+	-	+	+
SIGFPDIVIDE	44	MC68881 division by 0	+	+	-	+	+
SIGUNDERFLOW	45	MC68881 underflow	+	+	-	+	+
SIGOPERAND	46	MC68881 invalid operand	+	+	-	+	+
SIGOVERFLOW	47	MC68881 overflow	+	+	-	+	+
SIGSNAN	48	MC68881 signaling not-a-number	+	+	-	+	+
	49-63	Vendor-defined interrupts					

Notes: A = Default state is "abort" (otherwise, "ignore")
C = Interrupt can be caught
D = Produces a core dump
I = Interrupt can be ignored
R = Resets to default state when triggered
0 = See text

SIGTIME is not currently implemented.

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 shown in the table, 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 is written to a disk file immediately after receipt of the interrupt. The user can examine this file to determine the state of memory at the time the interrupt was received. A core file is often useful for diagnostic purposes. The operating system cannot create a core file if the working directory contains a file named "core" which denies write permission to the current effective user or if the working directory denies write permission to the current effective user.

The default action for the SIGDUMP interrupt is to create a core dump and return control to the task. The task is not terminated.

A vendor may use a TRAP instruction with a number greater than 6. In such a case the user should not issue the instruction.

User-defined interrupts are available to the end user.

Many of the interrupts are initiated by 68xxx exception processing. The cause of those interrupts can be understood by studying the documentation of the 68xxx microprocessor. Certain interrupts in the list are not directly initiated by the 68xxx and need further definition.

1) Hangup: Generated by UniFLEX when a terminal driver loses the carrier that it had previously established for modem operation. This interrupt causes the user associated with the terminal to be logged out. Certain programs (such as the text editor and BASIC) intercept this interrupt and take proper actions to save current files before logging out.

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

3) Quit: Generated by typing a control-\ on the terminal. This interrupt is just like the keyboard interrupt except that it produces a core dump.

4) A-line (Axxx) emulation trap: Generated by the 68xxx when it encounters an instruction with the pattern 1010 in bits 15 through 12.

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

6) Broken pipe: Generated when a pipe between two tasks is broken. This occurs when the reader has closed the pipe and the writer attempts to write to it.

8) Trace: An interrupt for use in tracing program execution.

9) Time limit: Generated when a task uses more system time than the upper limit established by the system configuration.

10) Alarm: Generated by the "alarm" system call after the specified number of seconds. Unless caught or ignored, this interrupt will terminate the task.

11) Task termination: The normal means of interrupting and terminating a task. Unlike the task-kill interrupt, the task-termination interrupt may be caught or ignored.

14) F-line (Fxxx) emulation trap: Generated by the 68xxx when it encounters an instruction with the pattern 1111 in bits 15 through 12.

21) Parity error: Generated when the system hardware detects a memory parity error. This interrupt is not supported on all hardware.

26) A child task terminated: 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.

27) Write to read-only memory: Generated when an attempt is made to write to a section of memory which has been reserved as read-only by the memory management system. This interrupt is not supported on all hardware.

28) Data or stack space violation: Generated when a program attempts to extend its stack below the limit specified in the most recent "stack_limit" system call. Also generated when a program attempts to execute a 68xxx instruction which resides in either the data or stack space. Instructions should only be executed from the text segment. This interrupt is not supported on all hardware.

29) Segmentation violation: Generated when an attempt is made to access memory which is outside the address space allotted to a task.

30-32) User-defined interrupts: These interrupts have no defined meaning to UniFLEX. They are merely additional interrupts which the end user may use for any purpose.

37-41) System-defined interrupts: These interrupts are reserved for use by Technical Systems Consultants.

42) MC68881 branch or set on unordered operand: Generated when an MC68881 branch or set instruction was executed with an operand equal to not-a-number (NAN).

43) MC68881 inexact result: Generated when either an MC68881 operation results in a loss of precision or the conversion of a number from packed-decimal to binary cannot be performed exactly.

44) MC68881 division by 0: Generated when an MC68881 divide instruction is executed with a divisor equal to 0.

45) MC68881 underflow: Generated when an MC68881 operation results in an answer that is nonzero but too small to be represented as a normalized floating-point number in the specified precision.

46) MC68881 invalid operand: Generated when an MC68881 operation is attempted on an operand that is outside of the domain of that operation (e.g., taking the square root of a negative number).

47) MC68881 overflow: Generated when an MC68881 operation results in an answer that is too large to be represented in the specified precision.

48) MC68881 signaling not-a-number (NAN): Generated when an MC68881 operation is attempted on an operand of a special type called "signaling NAN". The user must have supplied this operand because the MC68881 cannot generate a "signaling NAN" as the result of an operation.

49-63) Vendor-defined interrupts: These interrupts are reserved for use by the vendor.

On return from a "cpint" system call, the address which the system used on receipt of the program interrupt is returned to the 68xxx´s D0 register. This 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 a 1 returned in the D0 register. That 1 is the address which was previously being used on receipt of a quit interrupt, and we know that an address of 1 says to ignore the interrupt.

Knowing what type of action is being taken on receipt of a program interrupt 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 we have program "A" which starts program "B" by doing a "fork" system call followed by an "exec" system call. Also assume program "B" normally wishes to catch every keyboard interrupt (control-C) and process it in a special way. Program "B" should immediately check to see how program "A" was handling keyboard interrupts. If program "A" was not intercepting keyboard interrupts or was catching them, program "B" may catch and process them as desired. If, however, program "A" was ignoring keyboard interrupts, program "B" should also ignore them. The code for program B to handle all this properly would be:

```
              ...
              ...
              text
              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 having a keyboard interrupt to through during the checking and being improperly handled.

As an example of catching a program interrupt let's examine a portion of code that would put a program to sleep for 30 seconds. The technique will be to send an alarm interrupt with the "alarm" system call, then to 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.

```
              ...
              ...
              text
              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 which come in and handle them as specified by the code at "wake". The code at "wake" does absolutely nothing but return. Therefore, when the program receives the alarm interrupt it simply continues execution where it left off, which was just after the "stop" system call.

6.4.2 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 or not a program interrupt is pending. Once that system call is completed, the user´s program will see any waiting program interrupt. A few system calls, however, may be terminated by a program interrupt. Those calls are the "read" and "write" system calls (if the device being read from or written to is a slow device such as a terminal or printer) and the "stop" and "wait" system calls. A "read" or "write" call to a fast device, such as a disk file, cannot be terminated by a program interrupt. If a program interrupt does get through to a "read", "write", "stop", or "wait" system call, the following action takes place. First of all, the system call is immediately terminated and control passes to the program-interrupt handling-code if the interrupt is being caught. When the any special handling of the interrupt is complete, control passes to the instruction immediately following the interrupted system call and the system returns an error status accompanied by an EINTR error (number 27). In this way, the program which made the system call can detect that it was interrupted and reissue the system call if desired.

For 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 the "read" system call that gets the data from the terminal is in progress, that call may be prematurely terminated; that is, all the data may not 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. Alternatively, however, it could recognize the EINTR error and loop back in the code to reissue the prompt and the "read" system call.

6.5 Locking and Unlocking Records

In a multi-user, multi-tasking system such as UniFLEX, more than one user or task may attempt to access the same file at the same time. In some instances this is no cause for concern and, indeed, is often beneficial. Sometimes, however, it could be disastrous. For example, consider an airline reservation system. Assume one reservationist reads in a record containing information about the seats available on a particular flight and finds that only one seat is left. Normally the reservationist would type in the information necessary to reserve the seat, and when the transaction was complete, the system would rewrite that record to show that no more seats were available. A serious problem might result if another reservationist read the record after the first but before the first had written the new information out. Both reservationists would see the empty seat and enter the data to reserve it. There is no way of knowing just how the system would handle such a situation, but one passenger would probably not really be booked.

UniFLEX supports a method of avoiding this problem. It permits a program or task to "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 can access it. In our previous example of the airline reservations, the first reservationist would lock the record just before reading it and not unlock it until the new information had been written to the file. The second reservationist would be unable to read the record until the first reservationist unlocked it.

UniFLEX maintains a table showing what records are locked. The length of each locked record is specified by the task which performs the lock. Note that a single task may only lock one record in a file. However, other tasks may lock other records in that same file. A single task may lock a record in more than one file at a time.

The system calls which enable this feature are "lrec", which locks a record, and "urec", which unlocks a record. When a task issues an "lrec" call to lock some record within a file, the system first checks the locked-record table to see whether or not the calling task already has a record locked in that file. If it does, the system unlocks it. Next, the system checks to see if the record to be locked is available or if some other task has locked some portion of it. If the record is available for locking, the system makes an entry in the table of locked records and returns to the calling task. If the desired record overlaps some portion of a record that is already locked, the system returns with an ELOCK error. The calling program must decide how to handle such an error if it occurs.

A task may unlock a record it has locked in one of three ways. It may use the "urec" system call, which unlocks whatever record the calling program has locked in the specified file; it may lock another record in the same file, which automatically causes the system to unlock any record in that file which is currently locked by the task; or it may close a file, which causes the system to unlock any records that were locked by the task which opened the file.

Now that you understand how locking and unlocking take place, 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 assume the responsibility of avoiding reading from or writing to a locked record. This may be accomplished by attempting to lock a record before reading from or writing to it. If the record is available, the system does not return an error, and the program can safely proceed with a "read" or "write" system call. If the system does return an error (ELOCK), someone else already has the record locked, and the program should not try to read from or write to it at the present time. One way of handling an ELOCK error is to put the task to sleep for a few seconds before trying to lock the record again. This can be done with the "alarm" and "stop" system calls (see the example in Section 6.4.1 and the _Introduction to UniFLEX System Calls_). Proper use of the "lock" and "unlock" system calls yields the same results as if locking actually did prevent another task from reading or writing. Note that locking and unlocking are not necessary in all cases, only in those cases where a data file is shared and conflicts can occur.

An example of record locking and unlocking exists in UniFLEX BASIC. The BASIC interpreter always performs record locking and unlocking when accessing any record I/O file or virtual array (locking is not performed on sequential files). If one user does a "get record" in BASIC, that record is locked, and no other user may access it until the original user unlocks it by getting a different record, explicitly unlocks the record, or closes the file. Before any access to a random file, BASIC attempts to lock the record. If BASIC receives an error from the "lrec" system call because someone else has the record locked, it will pass that error to the user as BASIC error number 49.

6.6 Shared-Text Programs

In a multi-user environment more than one user may be running the same program simultaneously. Because it is wasteful of system resources for each user to have a copy of that program in memory at the same time, it makes good sense to let all the users share one copy of the program. This approach is only feasible, however, if nothing is ever altered in the memory which contained the program. Now, seldom does such a program exist, but an alternative is to separate a program into two portions, one of which contains code that will never be changed and one which consists of temporary storage and data that might require changes. Users can then share the nonchanging portion and have their own individual copies of the changing portion. This technique, which is implemented by UniFLEX, is called shared-text.

In UniFLEX all assembly language programs are separated into two sections: a "text" segment, which contains nonchanging memory or memory which will only be read from, and a "data" segment, which can be changed by writing into it. When a user runs a shared-text program, the system assigns a section of memory to each segment. If a second user runs the program at the same time, the system will recognize that it already has a copy of the text segment in memory and will load only the data segment into memory for the second user. The system will then map the same memory that contains the text segment for the first user into the address space for the second user.

For details on how to produce a shared-text program, refer to the manual 68xxx Relocating Assembler and Linking-Loader.

7.0 General Programming Practices

Several general programming practices should always be observed when writing assembly language programs to run under 68xxx UniFLEX. This section covers those points.

1) Starting Locations

Assembly language programs cannot have absolute origin addresses. Rather, the load addresses for the text and data sections of a program (as well as for the stack established by the system) should be specified at load time. The user can explicitly specify these addresses to the linking-loader, but in general they should assume the default values found in the file "/lib/std_env". This file, which contains the proper addresses for the hardware memory manager, is automatically read by the linking-loader.

2) Stack Considerations

When a program begins execution under UniFLEX, the system assigns it a portion of memory for the program stack. The 68xxx CPU's current 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 location of the current stack-pointer. The program may, of course, read the parameters which the system passes to it and which lie directly above the stack pointer (higher in memory).

3) 68xxx 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.

4) Delays

To maintain system efficiency, a user's program should not contain delay routines which tie up the processor for long periods of time. The preferred method is to use the "alarm" system call followed by a "stop" system call (see Section 6.4.1). The program must also use the "cpint" system call to catch the "alarm" interrupt and to continue with the desired code.

5) System "lib" Files Provided

The master UniFLEX disk contains several system library files for the convenience of the assembly language programmer. These files, which are located in the directory "/lib", contain definitions for several system-related calls, tables, buffers, and so forth. The programmer may include these definitions in a program by using the "lib" instruction in the 68xxx assembler. These files are as follows:

sysacct	Definition of structure for accounting record
sysdef	Definitions of 68xxx UniFLEX system calls
syserrors	Definitions of 68xxx UniFLEX system errors
sysfcntl	Definitions of interface for "fcntl" system call
sysints	Definitions of program interrupts
sysmessages	Definition of interface for intertask communication
syspty	Definition of pseudoterminal interface
sysrump	Definition of interface for resource manager
sysstat	Definition of buffer for "status" and "ofstat"
systim	Definition of buffers for "time" and "ttime"
systty	Definition of buffer for "ttyset" and "ttyget"
sys68881	Definition of the MC68881 exception buffer

The library also contains the following file, which is used by the linking-loader and which should not be included in an assembly language program:

std_env	Standard environment for linking-loader

6) Generating Unique Names for Files

It is often necessary for a program to generate a name for a file. For example, a program may need some sort of temporary file. In a single-task environment, the program could just use some name defined at assembly time. In a multi-user environment like UniFLEX, however, more caution is required. If the program which generates the name is run by more than one user or is run in the background and foreground by a single user, conflicts may arise because 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 in the name of the file. Because each executing copy of the program has a different task ID, they will each generate a different name for the file. The program should use the "gtid" system call to obtain the task ID, then convert the ID to ASCII and include it as part of the name of the file.

8.0 Debugging

Assembly language debugging under 68xxx UniFLEX is accomplished by the "qdb" command. This command provides many tools such as memory dumps, breakpointing, and single-stepping. Refer to the documentation on the "qdb" command for further details.

9.0 Sample UniFLEX Utility

To demonstrate several of the calls and techniques in writing assembly language utilities under UniFLEX, we shall provide the complete listing of a sample utility. This utility reads a file (or list of files) and strips out all control characters except for carriage returns (hexadecimal 0D) and horizontal tabs (hexadecimal 09). We will name the utility "strip". The syntax of the command line is as follows:

```
strip [<file_name_list>]
```

The square brackets indicate that the list of file names is optional. If the user does not supply the name of a file, "strip" will read standard input. If the user supplies a list of names, the "strip" utility will read all the files in order and write the stripped output to 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 of file, we shall read and write the data a buffer at a time. The question arises as to what size of buffer to use. We know that for efficiency reasons 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; the program and the buffer could easily fit into one 4-Kbyte page of memory (a common size for a page on a 68xxx memory management device). Since this utility will probably not be frequently used, it was decided to limit the program´s memory to one 4-Kbyte page. We will make the buffer for reading and writing the largest possible multiple of 512 that can fit in that 4-Kbyte space.

The printed listing of the "strip" utility follows shortly. It contains extensive comment and should be quite instructive in itself. We shall briefly talk through the code here, however, by referring to the line numbers printed on the left edge of the listing.

The first step after titling and describing the program is to include the system definitions with the "lib" instruction (line 17). The actual code section begins with the "text" statement in line 23. In line 27 we load the "a6" register with a pointer to the list of arguments passed to the program (the list is null if the user did not specify the name of a file). Notice that the program skips 8 bytes--4 containing the argument count and 4 containing argument 0, which is the name of the command itself. Lines 28 through 31 check to see whether or not the user specified any files 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 an argument. If the argument count is equal to 1, the user did not specify a file, and the program must read standard input. Because the file descriptor for standard input is 0, 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 0 (a null pointer), we have reached the end of the list and we jump off to the exit code at "done". If it is nonzero, it must be the address of a string designating a file. Lines 40 through 42 open that file for read and save the file descriptor in "ifd". Note that the open is done by means of an indirect system call because at the time of writing we do not know what name 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 pointer to the name of the file at "opname" in line 35, we were actually storing it 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 input file.

The "strip" subroutine is where the actual stripping of control characters takes place. In lines 67 through 69 we read "BUFSIZ" characters into memory at "buffer". Lines 73 and 74 check for end of file. If we are at the end of the file, we jump to the next occurrence of local label "90" and exit from the subroutine. Otherwise, we adjust the count for the "dbra" instruction (line 75) and go on to lines 80 through 91 where the control characters are stripped from the buffer. The reader should not necessarily be concerned with this routine except to note that after it strips the control characters, the program leaves the resulting data in the same buffer. Because some characters may have been stripped from the file, the location of the end of the data in the buffer may be lower than it was before the stripping. After the stripping, we fall into lines 96 through 101, which write the stripped data to standard output. Lines 96 and 97 calculate the number of characters to write. This number 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 as a parameter for an indirect "write" call. In line 98 we obtain the file descriptor for the standard output file. The indirect "write" system call is carried out in lines 99 and 100. In line 101 we jump back to the beginning of the subroutine to read in another buffer-full of data.

Lines 113 through 134 contain the error-handling code. On receipt of an error, we simply write an appropriate message to standard error (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 file descriptor of the current input file. Lines 155 through 158 reserve space for the buffer. As explained above, we decided to make the buffer as large a multiple of 512 bytes as possible and that will fit within 4 K. This is done by ensuring the buffer starts on a 512-byte boundary and then making the end of the buffer be the end of the 4-Kbyte page of memory. Recall that efficiency in reading and writing 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 4-Kbyte page ($1000) and the beginning of the buffer.

The "end" statement on line 161 specifies the starting address of the utility in its operand field.

The best way to learn to program is to program, so it is highly recommended that as a starting point the reader type in, assemble, and execute this utility.

```
*************************************************************
*
* UniFLEX "strip" Utility
*
* Copyright (c) 1984 by
* Technical Systems Consultants, Inc.
*
* Utility to strip all meaningless control characters from
* input file and write stripped version to standard output.
* Accepts list of input files or defaults to standard input.
* For the purpose of this utility, "meaningless control
* characters" are all characters with an ASCII value between
* $00 and $1f inclusive except carriage return ($0d) and
* horizontal tab ($09).
*************************************************************

         lib     sysdef         read system definitions

*******************************************
* start of main program
*******************************************

         text                   begin text segment

* start by seeing if any input files were specified

start    lea     8(a7),a6       set arg ptr past count & arg0
         cmp.l   #1,(a7)        file specified only if argcnt >1
         bhi.s   main2          branch if filenames present
         move.l  #0,ifd         else use standard input
         bra.s   main4          go process std. input

* check to see if any more files specified

main2    move.l  (a6)+,opname   get next argument in list
         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.l  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.l  #0,d0          show normal termination
         sys     term


```

```

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


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



* begin by reading a buffer full

strip    move.l  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.l   d0              end of input file?
         beq.s   90f             exit if so
         sub.w   #1,d0           adjust count for dbra
* 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.l  #buffer,a0      point to source buffer
         move.l  a0,a1           point a1 to destination buffer
         bra.s   60f             enter DBcc loop
40       move.b  (a0)+,d1        get a character into d1
         cmp.b   #$1f,d1         a control character?
         bhi.s   50f             go keep character if not
         cmp.b   #$0d,d1         a carriage return?
         beq.s   50f             keep if so
         cmp.b   #$09,d1         a tab?
         bne.s   60f             if not, don't keep
50       move.b  d1,(a1)+        put char. in buffer
60       dbra    d0,40b          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   wrterr          branch if error
         bra.s   strip           go read another section

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

* parameters for indirect "write" system call
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-(*-start_of_data)  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
```

Appendix A
Alphabetic Summary of UniFLEX System Calls

Call	No	Description	Syntax
alarm	43	Sleep for some seconds	Exp: <seconds> in D0 sys alarm Rtn: <previous_seconds> in D0
break	6	Change amount of memory	sys break,<high_address>
cdata	36	Request contiguous memory	sys cdata,<high_address>
chacc	25	Check access permission	sys chacc,<file_name>,<perm_mask>
chdir	21	Change directory	sys chdir,<dir_name>
chown	23	Change file owner	sys chown,<file_name>,<owner_ID>
chprm	24	Change access permission	sys chprm,<file_name>,<perm_mask>
close	15	Close file	Exp: <file_des> in D0 sys close
control_pty	65	Adjust or report the modes of a pseudoterminal	Exp: <file_des> in D0 sys control_pty,<function_code>,<mode_flag Rtn: <state> in D0
cpint	8	Catch program interrupt	sys cpint,<interrupt>,<address> Rtn: <old_address> in D0
create	11	Create a file	sys create,<file_name>,<perm_mask> Rtn: <file_des> in D0
create_ contiguous	61	Create a contiguous file	sys create_contiguous,<file_name>,<perm_ma <file_size>,<0_flag> Rtn: <file_des> in D0
create_pty	62	Create a pseudoterminal	sys create_pty Rtn: <slave_file_des> in D0 <master_file_des> in A0
crpipe	31	Create pipe	sys crpipe Rtn: <read_file_des> in D0 <write_file_des> in A0

crtsd	20	Make special file or directory	sys crtsd,<file_name>,<des_mask>,<address>
defacc	26	Set default access permission	sys defacc,<perm_mask>
dup	16	Duplicate open file	Exp: <file_des> in D0 sys dup Rtn: <new_file_des> in D0
dups	17	Duplicate specified file	Exp: <current_file_des> in D0 <requested_file_des> in A0 sys dups Rtn: <new_file_des> in D0
exec	2	Execute a program	sys exec,<file_name>,<arg_list>
exece	59	Execute a program with specified environment	sys exece,<file_name>,<arg_list>,<env_list>
filtim	52	Set file time	Exp: <time> in D0 sys filtim,<file_name>
fcntl	69	Change or query behavior of a file	Exp: <file_des> in D0 sys fcntl,<function_code>
fork	3	Fork a task	sys fork Rtn: new task starts just after call old task start at call + 2 bytes old task: <new_task's_ID> in D0
FPU_exception	67	Access or update FPU exception information	sys FPU_exception,<function_code>,<buf_add>
FPU_resume	68	Resume execution after an FPU exception	sys FPU_resume
gpid	60	Get parent task's ID	sys gpid Rtn: <parent_ID> in D0
gtid	32	Get task ID	sys gtid Rtn: <task_ID> in D0
guid	33	Get user ID	sys guid Rtn: <actual_user_ID> in D0 <effective_user_ID> in A0
nd	0	Indirect call	sys ind,<call>
ndx	1	Index indirect call	sys indx
ink	18	Link to file	sys link,<file_name_1>,<file_name_2>
ock	22	Lock task in memory	sys lock,<flag>

lrec	47	Lock record	Exp: <file_des> in D0 sys lrec,<count>
make_realtime	64	Make task "real time"	sys make_realtime
mount	29	Mount device	sys mount,<dev_name>,<dir_name>,<mode>
ofstat	27	Get open file status	Exp: <file_des> in D0 sys ofstat,<buf_add>
open	10	Open file	sys open,<file_name>,<mode> Rtn: <file_des> in D0
phys	54	Get physical resource	sys phys,<res_code>
profil	37	Profile task	sys profil,<start_add>,<buf_add>,<size>, <scale>
read	12	Read file	Exp: <file_des> in D0 sys read,<buf_add>,<count> Rtn: <bytes_read> in D0
rump	66	Resource management	Exp: <function_code> in D0 <resource_name> in A0 sys rump
sacct	50	Enable or disable system accounting	sys sacct,<file_name>
seek	14	Seek to file position	Exp: <file_des> in D0 sys seek,<position>,<mode> Rtn: <position> in D0
setpr	35	Set priority bias	Exp: <priority> in D0 sys setpr
spint	9	Send program interrupt	Exp: <task_ID> in D0 sys spint,<interrupt>
stack	7	Grow stack	Exp: <address> in A0 sys stack
status	28	Get file status	sys status,<file_name>,<buf_add>
stime	40	Set time	Exp: <time> in D0 sys stime
stop	44	Stop until interrupted	sys stop
suid	34	Set user ID	Exp: <user_ID> in D0 sys suid

term	5	Terminate task	Exp: <term_status> in D0 sys term
time	39	Get time	sys time,<buf_add>
truncate	55	Truncate file	Exp: <file_des> in D0 sys truncate
ttime	41	Get task time	sys ttime,<buf_add>
ttyget	45	Get terminal status	Exp: <file_des> in D0 sys ttyget,ttbuf
ttynum	51	Get terminal number	sys ttynum Rtn: <tty_num> in D0
ttyset	46	Set terminal status	Exp: <file_des> in D0 sys ttyset,<buf_add>
unlink	19	Unlink from file	sys unlink,<file_name>
unmnt	30	Unmount device	sys unmnt,<dev_name>
update	42	Update file systems	sys update
urec	48	Unlock record	Exp: <file_des> in D0 sys urec
vfork	56	Efficient fork on a virtual-memory system	sys vfork Rtn: new task starts just after call old task start at call + 2 bytes old task: <new_task's_ID> in D0
wait	4	Wait	sys wait Rtn: <task_ID> in D0 <term_status> in A0
write	13	Write file	Exp: <file_des> in D0 sys write,<buf_add>,<count> Rtn: <bytes_written> in D0