UniFLEX™ Programmer's Guide

Technical Systems Consultants, Inc.

UniFLEX™ Programmer's Guide

™ UniFLEX is a trademark of Technical Systems Consultants, Inc.

MANUAL REVISION HISTORY

Revision	Date	Change
A	7/81	Original Release
B	7/82	Corrected strip utility; added system calls.

DISCLAIMER

The supplied software is intended for use only as described in this manual. Use of undocumented features or parameters may cause unpredictable results for which Technical Systems Consultants, Inc. cannot assume responsibility. Although every effort has been made to make the supplied software and its documentation as accurate and functional as possible, Technical Systems Consultants, Inc. will not assume responsibility for any damages incurred or generated by such material. Technical Systems Consultants, Inc. reserves the right to make changes in such material at any time without notice.

TABLE OF CONTENTS

1.0 Introduction

The UniFLEX Programmer's Guide describes how to develop programs and utilities that execute under the UniFLEX Operating System. The assumed language in this manual is 6809 assembler; system programming in higher level languages will be discussed in separate manuals appropriate to those languages. This 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 assembler. Rather it is a brief and general introduction into writing simple utilities. The assumption is that the knowledgeable system programmer will be able to obtain the basics from this manual and then build on them with experience and the information in the UniFLEX System Calls manual to eventually develop utilities of most any complexity.

2.0 General

We begin with a general introduction into assembly language programs under UniFLEX: how they run, how they perform system function 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. The shell assumes the name which was typed is a file containing an executable binary program (there are execeptions such as command text files, precompiled BASIC programs, etc. 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, passing control back to the shell.

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 instance. 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 64K of address space. The task may not have a full 64K of RAM assigned to it, but it would have the entire 64K of address space available. Other tasks may be resident in other memory, but that memory is not mapped into the pu's 64K space. When the task terminates, its allocated memory is returned to the system and control is passed to the parent task (the task which created or initiated the terminating task).

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

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 system. The UniFLEX system or just "system" is essentially another task that is always available 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 id's, creating tasks, terminating tasks, mounting devices, reporting the time, and so on. 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 6809 UniFLEX Assembler.

2.2.1 The "sys" Instruction

The 6809 UniFLEX Assembler has a built-in instruction to make system calls to UniFLEX. It is the "sys" instruction and has the following format:

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

The only required portion of the operand is the <function>. It is simply a numeric code for the desired function. The parameters required depend on the particular function. There may be no parameters or as many as four. The function code is an 8-bit value while parameters are always 16-bit values. Many of the system functions also require certain values or parameters to be in one or more of the 6809 cpu's registers before executing a sys instruction. This depends on the particular function 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 function 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 functions 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. Under the USAGE section you will see the following:

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

This should be interpreted as follows. Before executing the read function call, the programmer must ensure that the desired "file descriptor" (we will find out what a file descriptor is later) must be loaded into the 6809's D register. Next we see the actual sys instruction and that in addition to the read function code itself, we must supply a buffer address (16-bit address of a buffer to read into) and a count (16-bit count of how many characters to read). After the read function has been executed, we see that the actual number of bytes read will be returned in the 6809's D register.

The actual system function code numbers can be found in a file of equates located in the "/lib" directory. This file is called "sysdef" for system call definitions. To see what a particular function code is, you could simply list the file with the command:

```
list /lib/sysdef
```

This file was provided on disk, however, so that the programmer could simply include those definitions in his program by including the sysdef file in his source via a "lib sysdef" instruction. APPENDIX A contains a printed alphabetical summary of the system calls along with their function codes, brief description, and calling syntax.

It is not necessary to know how the sys function works, but a brief description might help the programmer's overall understanding. The sys instruction generates a software interrupt. When this interrupt occurs, the handling routine maps the calling task out of the cpu's address space and maps the operating system code in. This system code then performs the requested function. It obtains the function number and parameters from the code directly following the software interrupt itself. When the system function has completed, the operating system is mapped out and the task is mapped back in to continue with its instructions.

2.2.2 System Call Example

Before cluttering things up, let's try a sample program. It will require the inclusion of a system function call. The simplest program we could write would be one which did nothing at all. As soon as it is initiated, it will immediately terminate. Thus the only system function we will need to call is the "term" function. Looking in the Introduction to UniFLEX System Calls manual, we see that there are no parameters required on the sys instruction itself (besides the function code), but that we must put a status value in the D register before performing the call. As the description states, if there are no errors this status should be zero. Thus we could write an extremely simple UniFLEX program that looked like the following:

```
        org   0
        lib   sysdef
start   ldd   #0          get status in d
        sys   term        terminate task
        end   start
```

All UniFLEX programs generally start at location 0, thus the "org 0" statement. The "lib sysdef" includes the definitions of all system function codes so that we can specify the term function as a symbol ("term") and not have to type in the particular number for that function. The "ldd #0" puts the status in D as required by the term function and then we have the actual system call to terminate. In the case of the term function, control is not returned to the calling program after execution of the call. Of course, that is the reason for the function; it terminates the current task (the task which made the call) and returns control to that task's parent. Notice that the program's end statement includes the symbol "start". This tells the assembler what the beginning location for execution is and also induces the assembler to make the resulting code executable by setting the permission bits.

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

```
++ asmb nothing.txt +ls +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. 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 may not be known at assembly time as they will be determined or changed during the execution of the program. Look at the read function 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 stuff the determined number into the appropriate bytes of the program following the software interrupt generated by the sys call. This is an unacceptable option since this would be self-modifying code. 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 and not program code. This allows the executing program to stuff parameters into those locations without creating self-modifying code.

The first of these indirect system call functions is called "ind". As described in the Introduction to UniFLEX System Calls, its format is:

```
sys ind,label
```

where "label" is the address of the memory locations that will contain the actual desired function code and parameters. Thus when this function is executed, the system goes to location "label" and there picks up the desired function code and any necessary parameters. That function is then executed and when 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 that number of characters to be 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 function call.

```
            ...
            ...
            ldd   rcount      pick up count to read
            std   iread+3     store in indirect read parameters
            ldd   fd          get file descriptor in d
            sys   ind,iread   do indirect read call
            ...
            ...
iread       fcb   read        setup read function code
            fdb   buffer      setup read buffer location
            fdb   0           space for read count (unknown)
buffer      rmb   $4000       reserve space for read buffer
            ...
            ...
```

At this point the reader should not concern himself with details of how the read 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" function. It is very similar to the ind function. The only difference is that instead of providing the parameter to ind which points to the parameters in memory, the indx function assumes that the pointer to the parameters in memory is in the X register. To see how this works, the above sample for ind could be repeated changing the instruction "sys ind,iread" to:

```
            ldx   #iread      get address of parameters in x
            sys   indx        do indirect read call
```

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

```
            ...
            ...
            ldy   rcount      pickup count to read
            ldx   #buffer     get read buffer location
            lda   #read       get read function code
            pshs  a,x,y       push on stack (order important)
            ldd   fd          get file descriptor in d
            leax  0,s         point to parameters on stack
            sys   indx        do indirect read call
            leas  5,s         clean parameters off stack
            ...
            ...
buffer      rmb   $4000       reserve 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 in the pshs instruction. Many 6809 programmers are under the mistaken assumption that the order that you specify the registers in the push instruction is the order in which they will be pushed. Not true! The 6809 has a fixed pushing and pulling order and if multiple registers are pushed or pulled in one instruction, this order is always followed regardless of how the registers were specified in the instruction. The above example forces the correct order by the choice of registers the parameters were loaded into. The 6809 pushes Y first, followed by X, and lastly A. Thus our parameters end up on the stack in the correct order. It is also important to note the "leas 5,s" instruction following the function call. It removes the parameters which were pushed onto the stack so that the stack is where it was before the system call section.

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 6809 condition code register. If the bit is zero on return, it implies that no error occured. If the bit is set (a one), then an error has occured and the D register contains an error number. The 6809 UniFLEX Assembler supports 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 Introduction to UniFLEX System Calls contains a list of the error numbers and their meanings. There is also a file of equates called "/lib/syserrors" which assign standard labels to the error numbers. These can be used in a program by simply including the file with a "lib syserrors" instruction. Note that UniFLEX does not report errors directly to the user. Error numbers are returned from system calls and it is entirely up to the user's program to report such errors or handle them as required by the specific application.

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. There may, of course, be several tasks "active" in the system at once. By active we mean they have been started and the system knows about them and is keeping track of them. There will be only one task or program actually executing at any given instant. Those tasks which are active but not executing at that instant are either mapped out of the cpu's available address space (64K) or are swapped out to disk. Now whenever that executing task performs some I/O or system call that will require him to wait, he 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 him to be mapped out, he will eventually run into a time-slice interrupt which will force him out so that other tasks can get some execution time. In this manner, multiple tasks can be run under UniFLEX at what seems like the same time. The switching of tasks occurs so rapidly that the user generally is not aware that he does not have the entire computer to himself. To assist in keeping track of all the active tasks, UniFLEX assigns a unique "task id" number to each task. This is a 15 bit unsigned value that can be used to uniquely identify a particular task. There is a system call named, "gtid", which a task or program can execute to obtain this task id if desired.

2.4.1 Address Space

To fully understand how the address space is allocated to a task, we must understand the technique of memory mapping which occurs in UniFLEX. Memory mapping is actually a hardware function that is under control of the software. Basically what happens is that the mapping allows the system to have more RAM memory than the cpu can directly address. The 6809 cpu can directly address only 64K bytes of RAM. With mapping we set up a physical address space that is much larger than 64K by adding additional address lines and registers. We look at this physical memory as a large pool of memory "pages". These "pages" are merely chunks of contiguous memory, generally 4K bytes in size. Memory mapping gives us the ability to select which of those pages of memory the cpu can directly access in its 64K address space. In other words, we "map" a portion of the large amount of physical memory into the 64K address space of the cpu. Thus when we wish to create a task, we can assign several of these pages into the 6809's 64K address space which is all that the computer and user program can directly address. The remaining unused pages are not mapped into the cpu's address space and are therefore inaccessible to the running task or program. Now if we decided to create another task, the pages assigned to the current running task can be effectively mapped out of the cpu address space so that we can map some other unused pages in for the new task. Note that the original task is still resident in RAM memory, it is merely inaccessible to the current task or program and must be mapped back into

the 64K address space (or just "mapped in") in order to run again.

Although the entire 64K address space of the cpu is available to a running task, there may not really be that much memory actually assigned or mapped into the space. In other words, a small task may only require 8K to run and the system can save memory by only mapping two pages into the 64K address space for this task. There would be no memory at locations above 8K in the cpu's address space. Indeed, this is exactly what UniFLEX does. UniFLEX allocates just enough memory for the program to run. The program memory starts at zero and runs upward. Generally all UniFLEX utilities and programs load and begin execution at memory location 0.

In reality, no single program can occupy the full 64K address space. This is because UniFLEX also allocates a page of memory at the top of the 64K address space for the program's stack. Thus the largest possible user program would be 60K of memory. When a task or program is initiated, the system stack pointer is left pointing to some location in the memory page which was allocated at the top of the address space. As we shall see in the next section, certain parameters are left on that stack which the task can make use of.

In short, when a task is initiated we know that it has 4K of memory at the top of the 64K address space for the system stack, and whatever number of pages or 4K chunks of memory required to contain the program starting at address 0. Later we shall see how to dynamically add more pages to the program or to the stack during execution if the address space is not already full.

2.4.2 Arguments

It is often desirable to pass arguments or parameters to a program when you begin its execution. UniFLEX allows for this in the "exec" system call. This is the 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 exec, but rather in 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 arguments are passed to a program by leaving them on the system stack. When the program is initiated, the 6809's system stack pointer 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 easily find these strings, we are also given a list of pointers to the beginning of the strings. In addition to that, we are also given

a count of how many arguments have been passed. This argument information is laid out as follows:

1) The stack pointer is pointing to the argument count. It is a 2 byte value and should always be greater than zero.
2) Just above the argument count (higher addresses in memory) is the list of pointers to the argument strings. These pointers are 16 bit addresses of the actual strings.
3) At the end of the pointer list are two bytes of zero to signify the end of the list. (A null pointer.)
4) The actual string arguments begin above the zero bytes. Each argument string is the string of characters that make up the argument followed by a zero byte.
5) Two additional bytes of zero are placed at the end of the argument strings to indicate the end of the argument data and signal the very top of the task's address space.

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, a file name, 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 name is "data2", and the option is "+b". Our argument count will be three. Let us arbitrarily say the system stack pointer is at $FDE0. We should see the following data on the stack:

	item	location	contents
	arg data terminator	$FDF8	$0000
	arg 2 terminator	$FDF7	$00
	argument 2	$FDF5	'+b'
	arg 1 terminator	$FDF4	$00
	argument 1	$FDEF	'data2'
	arg 0 terminator	$FDEE	$00
	argument 0	$FDEA	'pile'
	arg list terminator	$FDE8	$0000
	pointer to arg 2	$FDE6	$FDF5
	pointer to arg 1	$FDE4	$FDEF
	pointer to arg 0	$FDE2	$FDEA
sp ->	argument count	$FDE0	$0003

Thus if we wish to get the second argument (argument number 1), we read the pointer stored at the stack pointer + 4. That value is $FDEF. That is the pointer to the argument string itself and there we find the string of characters "data2" followed by a zero byte.

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

the command or program name itself. The arguments after the program name are then numbered sequentially beginning with one. 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 passes arguments to a command. In particular we are referring to name 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 name matching feature of the shell which allows the asterisk ("*") to match any string of characters. The interesting thing to note is how the list command actually knows to list those three files. The answer is that the shell does not pass "file*" as an argument to list but rather searches the directory for all filenames that match and passes them all as individual arguments. If the command line was as above, the list program would see four arguments:

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

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

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 function is executed, the task is halted and its memory is relinquished to the system. Before calling the term function, the programmer is required to place an error status value in the D register. When the task terminates, this value is passed back to the task's parent. If there is no error on termination, this error status should be zero to indicate a clean termination. If the task is terminating due to a system error such as an I/O error, the error value returned by that system call should be used as the error status for the term function. If the task 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 $00FF. 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 $00FF.

3.2 The "wait" System Call

The "wait" system function call is issued by a task when it wishes to wait for one of the children tasks it has spawned to terminate. It is through the wait command that the parent task receives the termination status from its child. The programmer issues the call with the following syntax:

```
sys wait
```

When the system call returns, the termination status is found in the X register and the terminated task's id is found in the D register.

If there are no children tasks when a wait call is issued, an error will be returned. If there is a child task that is still running when the parent issues a wait, 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, the system will save the child's task id and termination status until the parent does issue a wait. If several children tasks have been spawned, it is necessary for

the parent to issue a wait call for each one individually.

The termination status returned in the X register is comprised of two parts, the upper byte and the lower byte. The lower byte is the status value passed by the "term" system call. If this byte is non-zero, there was some sort of error condition which caused termination. Under normal termination conditions, the high byte of the termination status is zero. If non-zero, it indicates that the task was terminated by some system interrupt and the least significant seven bits of this byte contain the interrupt number. If the most significant bit of this byte is set, a core dump was produced as a result of the termination. Interrupt numbers and core dumps 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's prompt. The shell 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. The tool used to load and execute another program or binary file is the exec system function. That is the very function which the shell uses when it loads and executes a program (remember the shell itself is just another program). The program which makes the exec call is thrown away and the new program (a binary file) is loaded into memory and executed. The same task id number is retained. If the exec is successful (ie. no errors such as the file not existing), there will be no return to the calling program. Recall that the calling program is thrown away, making it impossible to return. If, however, there is an error in attempting to perform the exec function, the system will not load the new program but will return an error status to the calling program which is still intact. Thus a properly written program will 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 format is as follows:

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

The "fname" is the pointer to the filename. This filename is a string of appropriate characters somewhere in memory which is terminated by a zero byte. The "arglist" is the pointer to a list of argument pointers. In other words, "arglist" is an address at which we will find a list of pointers stored. This list of pointers is consecutive 2-byte addresses or pointers to the actual argument strings. The list is terminated by two bytes of zero which could be considered a pointer to zero. Each pointer in the list is the address of the actual argument string which is terminated by a zero byte. When the exec function is complete, the new program will have these arguments available in the exact format described in section 2.4.2 above.

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

```
        org   0
        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.
* Note the d register already has the error from exec.

        sys   term

* strings and data

filen   fcc   '/bin/dir',0
arg0    fcc   'dir',0
arg1    fcc   '+ba',0
args    fdb   arg0,arg1,0

        end   start
```

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

3.4 The "fork" System Call

The "fork" system call is used to spawn a new task. It is the only way to create a new task under UniFLEX. When the fork function is invoked, it creates a new task which is almost identical to the old (the old task is still around). This new task has the same memory and stack allocation, same code in the memory space, same open files, pointers, etc. Immediately after a fork you essentially have two identical tasks or programs running on the system. Now usually you want the new task to do something different, so 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 to start background jobs. When the shell sees a command ending with an ampersand ("&"), instead of directly doing an exec it does a fork to create a second shell. Now the newly created shell will do an exec of the desired command while the old shell is still around to accept further commands.

The syntax of the fork command is simply:

```
sys fork
```

There are no parameters required for the fork call. 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 in the instruction immediately following the system call to fork. The old task resumes execution at a point two bytes past the system call. In this manner, the same program can be run in two tasks via a fork and yet do different things after the fork. Since the new task resumes directly after the fork call and the old task resumes two bytes after the fork call, it is obvious that the first instruction in the new task must be a short branch instruction (requires 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 D register upon return from the fork. If an error occurs when attempting a fork, the new task will not be created and an error status will be returned 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   newtsk          branch to code for new task
    * old task resumes execution here
            lbes  frkerr          check for error, branch if so
            shs   d               save new task's id
    prwait  sys   wait            wait for child task
            cmpd  0,s             right one?
            bne   prwait          wait some more if not
            puls  d               restore child task id
            ...                   continue code for old task
            ...
            sys   term
    newtsk  sys   exec,name,args  new task probably does exec
            lbra  excerr          branch if error in exec
            ...
             ..
```

In this example, the old task waits for the new one (it's 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 6809 D 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 that the system programmer has a very good grasp on 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 the same. 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 concept is termed "device independent I/O". It means you can develop a program that sends its output to a terminal and that same program, without change, will also be able to output to a disk file, printer spooler, pipe, or any other device on the system. This feature lends a great amount of versatility to the system and makes program development and updating much smoother.

This device independence is made possible by the device driver routines. These are the system routines that take care of the specifics of the device for which they are written, creating a standard interface to the device. There is a routine to open the device and one to close it. These permit the system to do anything necessary to 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 or write data from the device.

4.1.2 File Descriptors

When a user wishes to perform some operation on a UniFLEX file, he informs the system which file he wishes 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 2-byte numeric representation of a specific file or device. This file descriptor number is assigned to the file by the system when that file is opened or created. UniFLEX then keeps track of the file descriptors and to which files they are assigned. In this way, the user need only supply a number instead of an entire file name each time the file is to be referenced.

As an example, look back at the description of the read system call to UniFLEX. You will see that this function requires a file descriptor value in the D 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 D register with that number.

File descriptor numbers begin with 0 and extend up to the maximum possible number of open files on the system. This maximum will vary depending on the system configuration, but generally will be around 12-25 for 6809 UniFLEX.

4.1.3 Standard Input and Output

When the shell begins execution of a task, it automatically assigns input and output files to that task. Generally the input file is the user's keyboard and the output file is the user's display. In fact, when a task begins execution it can always count on three input/output files being already opened, assigned a file descriptor, and ready for reading or writing. These three are called "standard input", "standard output", and "standard error output". The standard input is an open file ready for reading and always assigned a file descriptor of 0. Generally the standard input file is the user keyboard. Standard output is an open file ready for writing to and always assigned a file descriptor of 1. Generally the standard output file is the user display device. The standard error output is an open file ready for writing to and always assigned a file descriptor of 2. This output file is reserved for reporting error messages. It is almost always the user display device. The programmer can assume a message sent to standard error output will be delivered to the user's display.

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

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

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 input and output to a file, it should generally close that file. A user 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. This procedure is required no matter what the file. The format of an open system call is:

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

The "fname" is a pointer to a zero terminated string containing the name of the file to be opened. The "mode" is a number (0, 1, or 2) which sets the read/write mode. If 0, the file is opened for reading only. If 1, the file is opened for writing only. If 2, the file is opened for both reading and writing. On return from the open call, the 6809's D register will contain the 2-byte file descriptor number assigned to that file. All future references to the file will be made via this file descriptor. An error will be returned from this call if the file to be opened does not exist, if the task opening the file does not have proper permissions, if too many files are already opened, or if the directory path leading to the file cannot be searched.

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. First is that since the system has a finite number of files which may be open at one time, closing a file will free up a slot in which another file may be opened. The second reason is that in the case 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 D register and 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, etc. other system calls must be used. At this point we are only interested in creating disk files. The format of create is:

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

Once again, "fname" is a pointer to a zero terminated string containing the name of the file we wish to create. The file will be created in the default directory unless a particular directory is explicitly specified in the file name. The "perm" is a value which permits the user to set the desired permissions on the new file. The reader should refer to the Introduction to UniFLEX System Calls for details of setting these permissions. Note that if the file already exists in the specified directory, it will be truncated to zero length (all existing data deleted). In addition, the original permissions will be retained regardless of the "perm" value supplied to the create call. In other words if the file "fname" already exists, the "perm" parameter on the create call will be ignored.

If the file does not exist, permission setting will be subject to any default permission settings the owner of the file has previously specified. A brief description of default permissions follows in the succeeding paragraph. The "perm" parameter in the create call will allow you to deny permissions which the default permissions grant, but will not allow you to grant permissions that the default permissions deny. This can be thought of as a logical AND of the "perm" parameter and the default permission byte.

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

4.3 Reading and Writing

Perhaps the most heavily used system calls are read and write. It is by these functions that a program communicates with the user, disk files, printers, other tasks, and anything else in the outside world. It is imperative that the would-be UniFLEX system programmer get a good grasp on these calls, their use, requirements, and characteristics. Reading and writing under UniFLEX is very elementary and as such permits great versatility in how 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. This makes both sequential and random access of the files quite simple.

The read and write system calls assume a "file position pointer" has already been set. This is a pointer which the system maintains to show the current position for reading and writing in a file. We will see how it can be set in the section on "seeking". The only parameters required, then, are the file descriptor to specify which file, the count of characters to be read or written, and a memory buffer address to read into or write from. We shall look at each call separately.

4.3.1 The "read" System Call

To execute a read call, the programmer must first load the D register with the file descriptor number. Then he makes the call with the following syntax:

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

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

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

Reading and writing to terminals is handled in the same manner as reading and writing disk files. Identical system calls are used for terminals or disk files. There is a difference in the result of a read call, however, in that if the file being read is a terminal, only one line will be returned at most. By one line we mean all the characters typed since the last carriage return, terminated by a carriage return. Thus even though we execute a call with a desired "count" of 1024 characters to be read, if the user at the terminal types the letters "halt" followed by a carriage return, the read call would return with an actual bytes read count of only five. If the user has not typed anything when the call is issued, the calling program 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 and receiving no error and no characters. An "end of file" condition from a terminal is produced by typing a CTRL D. Note that the CTRL D character itself is not actually passed on 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 1024 bytes of data, placing them in a buffer at location $2000. We assume the file has already been opened for read and the file descriptor is stored at "fdsave".

```
            ...
            ...
            ldd   fdsave              get file descriptor
            sys   read,buffer,1024    read 1024 bytes into buffer
            lbes  rderr               branch if error
            cmpd  #0                  end of file condition?
            lbeq  endof               special handling if so
            addd  #buffer             point to end of data
            std   bufend              save buffer end pointer
            ...
            ...
            org   $2000
buffer      rmb   1024
            ...
            ...
```

Upon return from the "read" system call, we first check for a returned error status. If there was an error we assume the program handles it properly at "rderr". If no error, we check for an "end of file" condition. Recall that an "end of file" condition is recognized by a program as zero characters read when there was no error. If we are at the end of the file, the program jumps to "endof" where we again assume such a condition is properly handled. If we did not receive an error and were not at the end of the file, our program calculates a pointer to one past the last byte read into the buffer and stores that pointer at "bufend". Normally this pointer should be $2400 ($2000+1024), but if the read call returned less than 1024 bytes it would be lower.

4.3.2 The "write" System Call

The write function is executed by first loading the D register with the file descriptor number and then issuing a statement of the form:

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

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

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

```
            org   0
            lib   sysdef        include system definitions

      * start of main program

      sayhi   ldd   #1          write to std. output
              sys   write,hello,hlng  send message
              bec   done        exit if no error
              pshs  d           else, save error number
              ldd   #2          write to std. error output
              sys   write,erm,elng  sen error message
              puls  d           restore error number
              bra   done2
      done    ldd   #0
      done2   sys   term        terminate program

      * strings

      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
```

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

4.3.3 Efficiency in Reading and Writing

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

By all means do not perform single character I/O with system calls for each character. If single character I/O is required, the user program should handling the necessary buffering such that system calls are made only on a buffer full of characters. A sample set of good "character at a time" I/O routines are included in APPENDIX B.

4.4 Seeking

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

```
sys seek,offsethi,offsetlo,type
```

Before making a system call to seek, the user must load the desired file descriptor in the D register. Seeks are done on a relative basis. That is, a seek amount is supplied to the call and the seek is to be that amount relative to some reference point. This reference point must also be specified and is the "type" parameter shown above. There are three possible reference points: the beginning of the file, the current position in the file, and the end of the file. The "type" value should be as follows:

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

The arguments "offsethi" and "offsetlo" represent the high and low order portions of a four byte 2's complement offset. This is the amount of offset to be added to the reference point to find the new position in the file. A positive number indicates forward in the file while a negative number indicates backward into the file. On return from the

seek call, the new current position is found in the X and D registers (high order portion in X). This is the current position relative to the start of the file. To find the current position in a file, you could use a system call of "sys seek,0,0,1", finding the result in X and D.

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 200 records in our file, so the record number can be represented in 8 bits. We wish to write a subroutine which will read the record specified by the record number in the a register and leave the data at the location specified by the X register. The basic procedure will be to find the starting position of the desired record in the file by multiplying the record number by the record size of 256. Then we seek to that position and read 256 bytes. Our routine looks like this:

```
          ...
          ...
getrec    pshs  x          save address for data

* Since the record number is in a, we can get 256 times
* it by clearing b (equivalent to shifting no. left 8).

          clrb             record no. * 256 into d

* seek to desired record

          clr   iseek+1    setup seek address parameter
          clr   iseek+2
          std   iseek+3
          ldd   fd         assume file descriptor at fd
          sys   ind,iseek  indirect call to seek
          lbes  skerr      branch if error

* file pointer positioned, now read record

          puls  x          get address for record data
          stx   iread+1    put in read call parameters
          ldd   fd         get file descriptor
          sys   ind,iread  indirect call to read
          lbes  rderr      branch if error
          .ts              all finished

          ...
          ...
iseek     fcb   seek       seek function code
          fdb   0          high portion of address
          fdb   0          low portion of address
          fdb   0          type 0: position from begin
iread     fcb   read       read function code
          fdb   0          buffer location
          fdb   256        character count to read
          ...
          ...
```

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

4.5 File Status Information

Certain information about each file or device is available to the user. This information is obtained via the "status" or "ofstat" system call. The two calls differ in that "ofstat" is used to obtain information about a previously opened file while "status" obtains information from an unopened file. The format for ofstat is:

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

Note that the user must load the D register with the file descriptor of the previously opened file. The format for status is:

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

With status, the file is specified by providing the "fname" parameter which is a pointer to a zero terminated string containing the desired file name. In both commands the "buffer" parameter is a pointer to a buffer in memory or an area of memory into which the information about the file can be placed. This buffer must be at least 21 bytes long. When the status or ofstat call is completed, this buffer will contain all the information available for the file in the format described below.

Assuming the buffer begins at some location called "buf", the information in the buffer is as follows:

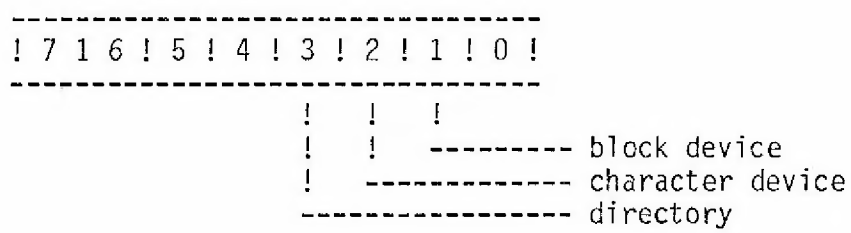
name	location	field size	information in field
st_dev	buf	2	device number
st_fdn	buf+2	2	fdn number
st_mod	buf+4	1	file mode
st_prm	buf+5	1	permission bits
st_cnt	buf+6	1	link count
st_own	buf+7	2	file owner's user id
st_siz	buf+9	4	file size in bytes
st_mtm	buf+13	4	time of last file modification
st_spr	buf+17	4	spare - for future use only

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. This is a block of information about the file and

where it resides on the disk. It is from the fdn that status and ofstat obtain their information. The file mode and permissions will be explained in the next paragraph. The link count is the number of directory entries that are linked to the fdn or actual file. More information on linking can be found later in this manual in the section titled "Directories". The file owner's user id is a two byte id that was assigned to the user by the system manager when he was given a user name. The file size in bytes is the exact number of characters in the file. The time of last modification is the internal UniFLEX representation of the last time the file was written to.

The file mode and permission bytes each hold several bits of information. This is done by assigning single bits within the file mode to particular file types and within the permission byte to the various possible permission types. The state of the particular bit (0 or 1) indicates which type of file mode or whether permission is given or denied. The file mode looks like this:

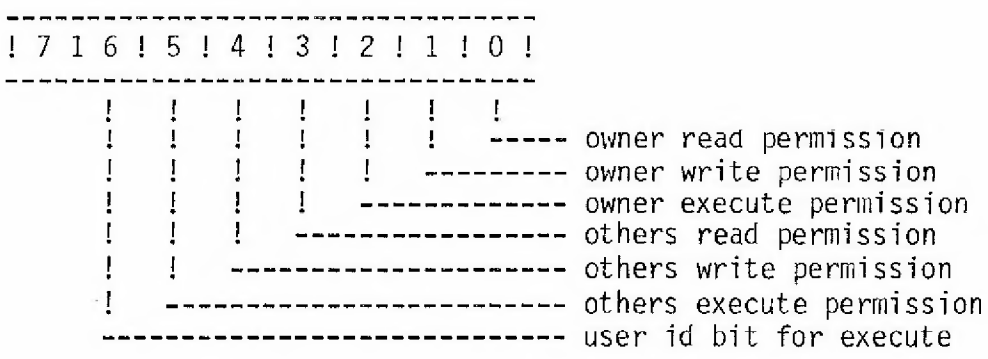
file mode (st_mod):



Notice that only three bits are used in this byte. Only one of the three bits should be set at a time and it indicates which type file this is. If the file is a regular disk file, none of the bits will be set. A block device is a device which handles data in 512 byte blocks such as a floppy disk drive. A character device is one which handles data single character at a time such as a terminal.

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

permissions (st_prm):



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

cleared, permission is denied.

The "user id" permission bit requires further clarification. If this bit is set, it gives the user of a file the same permissions as the owner while that file is executing. As an example of the usefulness of this feature, consider a user, "joe", who has a database program which manipulates a large data file. Now "joe" does not want anybody on the system to be able to directly read or write his data file, so he denies read and write permissions 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 for other users to be able to run his database program which manipulates the data file. All he needs do is set the "user id" permission bit in his database program. With the "user id" bit set, anyone who runs the database program has the same permissions as "joe" which allows them to manipulate the data file while running the database program. As soon as the database program is terminated, 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 available to all users under UniFLEX. A directory is a special type of file, and the only way to create a directory is with the "crtsd" system call. That call may only be executed by the system manager. Without the "user id" bit, the only person who could use the "crdir" command (which contains a "crtsd" system call) would be the system manager. The "crdir" program has the "user id" bit set, however, so that anyone who runs it temporarily has the same permissions as the owner. The owner of "crdir" is the system manager, so any user is thus capable of creating a directory.

5.0 Directories and Linking

A UniFLEX directory entry is nothing more than the name of the file and a single pointer. This pointer points to the file descriptor node or "fdn" for the file. This fdn is a small unit on the disk which contains various information about a particular file. There is one and only one fdn on a disk for each file which resides on the same disk. It is possible, however, to have more than one directory entry point to the same fdn. Two different users could have an entry in their own directory which pointed to the same fdn and therefore the same file. This feature is called a "link" and you can see it is possible to have many "links" to the same file. If you do a long directory listing on a directory (dir +l), you will find one entry for each file in that listing which is the link count. That is the number of directory entries which point to or are linked to the file. This count should be one at a minimum; if it ever goes to zero that means no one is linked to the file and it will be deleted. In fact when you "kill" a file, the kill command merely removes that name from the directory. This decrements the link count in the associated fdn. If that count is still non-zero, someone else is linked to the file and it is not deleted from the disk. If the count does go to zero, no one else is linked to the file and it is deleted.

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

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

If a file is open at the time an unlink call is made, the unlink will take place, but the file will not be deleted or closed by the operation. The user can still read or write to the file as long as it is left open. UniFLEX waits until the file is actually closed and then checks the link

count to see if it should be deleted from the disk. This creates interesting possibilities for a program. A file can be opened and then immediately unlinked. Now as long as the program leaves that file open, it can read from it or write to it. When the program is finished with the file, it has only to close it. If no one else is linked to the file, it will be immediately deleted.

6.0 Other System Functions

This section is devoted to several features and functions available to the system programmer that are somewhat specialized. Specific calling formats and parameters will not always be given. It is assumed that the reader should be able to obtain this information from the Introduction to UniFLEX System Calls.

6.1 The "break" and "stack" Functions

Earlier we learned that when a task is started, it is allocated memory according to the program size. Additionally, each new task is allocated one page of memory (usually 4K) for program stack space. If the total program and stack space allocated is less than 64K, we have a "hole" or unused address space between program and stack. It is possible for a running task to change the amount of memory allocated to it. Additional memory can be requested for the program space and for the stack space. It is also possible to relinquish allocated memory back to the system, that is to deallocate program memory or stack memory. The means of performing this dynamic memory or stack allocation and deallocation are the "break" and "stack" commands. An address is supplied to break and the system attempts to allocate memory to be sure there is RAM from location 0 up through the specified address. Memory is allocated in pages (4K blocks), so depending on the address specified there may be some memory beyond the address. If an address is specified which falls below the amount of program memory already allocated, that memory is relinquished or returned back to the system. The "stack" command works in much the same way, except that it grows downward in the cpu's address space.

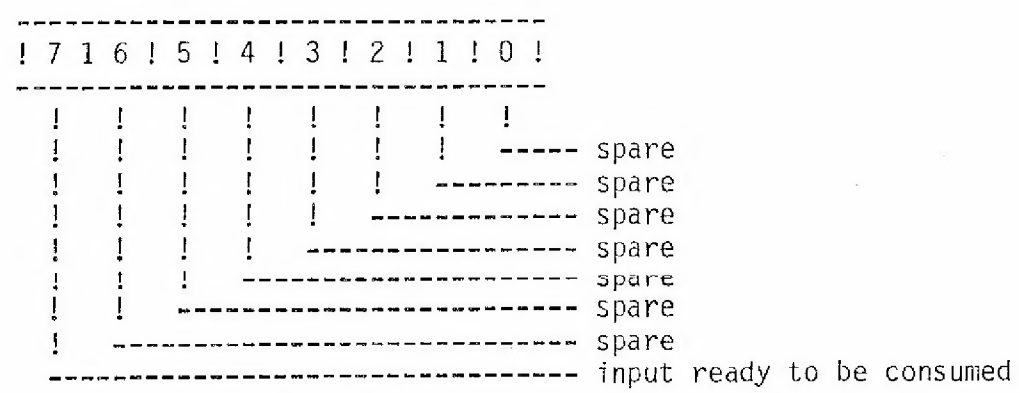
6.2 The "ttyset" and "ttyget" Functions

It is possible to alter and examine several configuration parameters of terminals under UniFLEX. These parameters include such things as the line cancel character, the backspace character, adjustable delay after carriage returns, mapping of upper to lower case, tab expansion, etc. The configuration of all these parameters is represented in six bytes of data. These six bytes can be read with the "ttyget" system call to examine the current configurations or can be set with the "ttyset" system call to alter the current configuration. A six byte buffer must be established in memory which contains the desired configurations for ttyset or will receive the current configuration information for ttyget. If we assume this buffer begins at "ttbuf", the data has the following format:

name	location	contents
tt_flg	ttbuf	flag byte (described later)
tt_dly	ttbuf+1	delay byte (described later)
tt_cnc	ttbuf+2	line cancel character (default is CTRL X)
tt_bks	ttbuf+3	backspace character (default is CTRL H)
tt_spd	ttbuf+4	terminal speed (described later)
tt_spr	ttbuf+5	reserved for future use

The terminal speed byte presently implements only one bit. It is the high order bit (bit 7) and if set indicates that the terminal has input characters waiting to be consumed by the program. This bit is only meaningful when read, ie. the input ready condition cannot be set via this bit and ttyset. The byte looks like this:

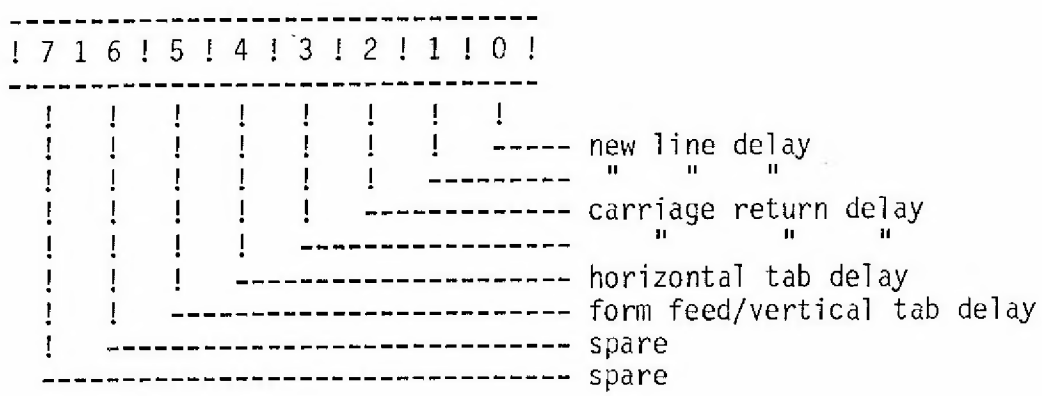
terminal speed byte (tt_spd):



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. There are special input modes which can be established, however, where the "input ready to be consumed" bit will come on as soon as a single character is input. These are called the "raw I/O mode" and the "single character input mode" as described later.

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

delay byte (tt_dly):



The new line delay is performed after each "line feed" character is output ($0A). It may be a zero delay or three actual delay times as shown here:

bit 1	bit 0	delay amount in milliseconds
0	0	0
0	1	10
1	0	20
1	1	30

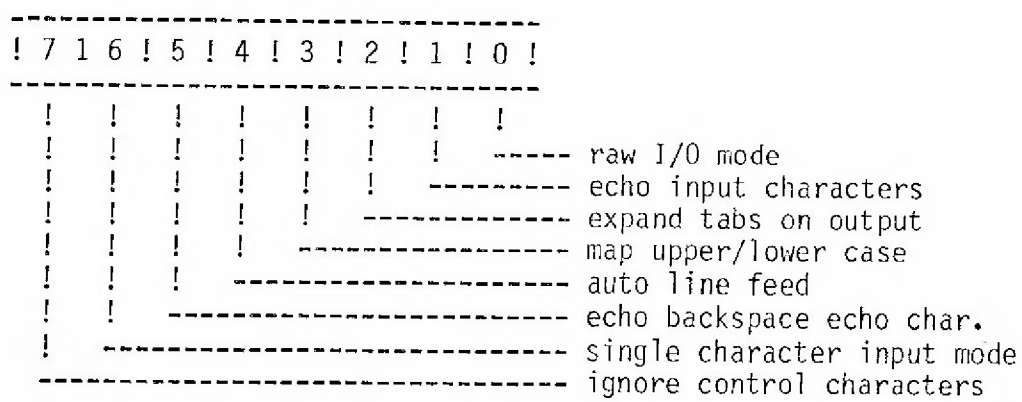
The carriage return delay is performed after each "carriage return" character is output ($0D). It may be a zero delay or three actual delay times as shown here:

bit 3	bit 2	delay amount 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 amount is 20 milliseconds. The form feed or vertical tab delay may also be either on or off. If on (bit 5 set), the delay amount is 240 milliseconds.

The eight bits of the flag byte represent eight different modes of operation for the terminal. When set they imply that the indicated mode is in operation. The format is as follows:

flag byte (tt_flg):



We shall describe each of these modes separately.

0) Raw I/O Mode

When in the "raw mode", the terminal drivers effectively do no special processing of the input or output characters whatsoever. Each and every character typed on the terminal is directly input to UniFLEX including backspace characters, line cancel characters, tab characters, CTRL C characters, and so on. Similarly, every character output to the terminal is output directly: no tab expansion is performed, no line feeds are appended to carriage returns, etc. In addition, the parity bit is not stripped on either input or output. When in the raw mode, the executing program has complete control of every character input or output and must perform any special processing itself. Under raw mode a read system call will not have to wait for an entire line to be input before it can read characters. If there is a single character available, the read call 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.

1) Echo Input Characters

If this mode is enabled, each character typed on the terminal will be echoed to the display device. In such a case, the terminal should be operating in full-duplex. An example of the use of this mode can be seen when a user logs in and is asked for his password. The login program writes the "Password:" message and then turns the "echo input characters" bit off while the password is entered. In that way the password is not echoed to the screen. This mode is on by default.

2) Expand Tabs on Output

If the terminal does not have hardware tab expansion, this bit can be set to allow the terminal driver software to automatically expand tabs on output. Tab stops are assumed to be at 8 column intervals. In other words, if this bit is on, each time a horizontal tab character ($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). This mode is on by default.

3) Map Upper/Lower Case

When using UniFLEX, it is generally assumed that the terminal has upper and lower case capability and that the user will type most commands and input in lower case. It is possible, however, to use an "upper case only" terminal by instructing the terminal drivers to map all typed input characters from upper to lower case and to map all output characters from lower to upper case. This is done by turning on the "map upper/lower case" bit in the ttyset flag byte. When this mode is on, mapping is done in both directions. By default this mode is off (assumes lower case capability). It is automatically turned on, however, if a user logs in with upper case characters for his name. In this way an "upper case only" terminal can be connected to UniFLEX without special considerations.

4) Auto Line Feed

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

5) Echo Backspace Echo Character

If this mode is on and if the backspace character is defined to be a CTRL H ($08), the terminal drivers will echo the CTRL H, then output a space, and then output another CTRL H. This will erase the incorrect character for terminals which do not do so automatically. This mode is on by default.

6) Single Character Input Mode

There are certain applications where a program might desire to input one character at a time without having to wait for a carriage return. That is the function of the "single character input mode". When not in the single character input mode, a call to read a single character would have to wait until an entire line terminated by a carriage return had been typed before it would have access to a single character within the line. If the single character input mode is on, the program can read a character as soon as it has been typed without having to wait on an entire line and carriage return. Note that it is still possible to read multiple characters while in the single character input mode, if they are available. While in the single character input mode, the parity bit is stripped off of input characters but only CTRL C, CTRL D, and CTRL Backslash are treated as special characters. In other words, tabs,

backspaces, and line cancels are ignored and should be processed by the user's program if desired. This mode is off by default.

7) Ignore Control Characters

When this mode is on, UniFLEX will ignore all control characters which do not have special meaning. In other words, all control characters will be ignored except for the following:

Carriage Return
Horizontal Tab
CTRL C
CTRL D
CTRL Backslash
Backspace Character (if defined to be a control char.)
Line Cancel Character (if defined to be a control char.)

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

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

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

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

```
            ...
            ...
            sys   crpipe    create pipe system call
            lbes  piperr    branch if error
            std   rdfd      save read file descriptor
            stx   wrtfd     save write file descriptor
            sys   fork      fork to spawn task B
            bra   child     new task B here
            lbes  frkerr    task A checks for error
            std   tskBid    save task id of child
            ldd   rdfd      pipe read file descriptor
            sys   close     close read (A only writes)
            ldd   wrtfd     pipe write file descriptor
            std   pipefd    save as pipe file dscrptr.
    * now task A can write to pipe using pipefd
            ...
            ...
            sys   term      end of task A

    * code for task B

    child   ldd   wrtfd     pipe write file descriptor
            sys   close     close write (B only reads)
            ldd   rdfd      pipe read file descriptor
            std   pipefd    save as pipe file dscrptr.
    * now task B can read from pipe using pipefd
            ...
            ...
```

There is one major point to learn from the above example. That 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. After performing the fork, both tasks have open read and write pipe files. Now it is assumed that the writing task will eventually close the write pipe file and the reading task will eventually close the read pipe file, but we must specifically be sure that the writing task closes the read file and the reading task closes the write file. In fact, these closes should be performed as soon as possible, before any reads or writes to the pipe are performed.

6.4 Program Interrupts

UniFLEX provides a mechanism to interrupt tasks under software control. These interrupts are called "program interrupts". It is possible for one program or task to send one of these program interrupts to another task. This permits timing and synchronization among the tasks in the system. It also gives the programmer the ability to terminate tasks prematurely under software control.

6.4.1 Sending and Catching Program Interrupts

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

```
...
...
ldd   #327          get task number in d
sys   spint,QUITI   send quit interrupt
lbes  error
...
...
```

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

Often it is possible for a task to "catch" or intercept a program interrupt when it is received. The task may then permit the interrupt to complete its default action (usually task termination), may ignore the interrupt completely, or may take some special user-defined action. The system call that provides this ability to catch a program interrupt is called "cpint". In effect, this call permits the user to setup an interrupt vector address such that if a program interrupt is received, control is vectored to that address. The programmer may place a routine at that address which handles the interrupt in some special way. Two addresses are special, $0000 and $0001. If the address specified for the caught interrupt is $0000, 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 $0001, the interrupt will be ignored much as if the interrupt had not even been sent. Note that no code is actually placed at these addresses. The cpint function recognizes them as special values and performs the indicated interrupt handling without ever jumping to or using them as real addresses. Any other address supplied to cpint is assumed to be a valid program address in memory and control is passed to that location. There the programmer places the desired interrupt handling routine which must be exited with an RTI instruction. When this RTI is executed, control is resumed at the same point in the program where the interrupt occured.

Once a program interrupt has been caught and processed, the system resets itself back to the default condition such that the interrupts are no longer intercepted. Therefore, to continue catching program interrupts it is necessary to re-issue a new "cpint" call after each interrupt is processed.

Following is a list of the types of program interrupts possible under UniFLEX.

name	number	description	comments
HANGI	1	hangup interrupt	
INTI	2	keyboard interrupt	
QUITI	3	quit interrupt	produces core dump
EMTI	4	emt interrupt (swi)	produces core dump
KILLI	5	task kill interrupt	can't be caught/ignored
WPIPI	6	write broken pipe int.	
BARGI	7	bad argument interrupt	produces core dump
TRACI	8	trace interrupt	produces core dump
TIMEI	9	time limit interrupt	produces core dump
ALRMI	10	alarm interrupt	
TERMI	11	task termination interrupt	
USERI	12	user-defined interrupt	

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

A description of each program interrupt follows. All of these interrupts may be produced via the "spint" system call, but some are automatically generated as documented below.

1) Hangup Interrupt: 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 automatically logged off. Certain programs (such as the editor and BASIC) intercept this interrupt and take proper actions to save current files before logging off.

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

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

4) *EMT Interrupt (swi): Generated by a "swi" software interrupt in an* executing program. May be used for debugging purposes.

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

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

7) Bad Argument Interrupt: Can be generated by the shell if the arguments provided are illegal. At the time of this writing, no software specifically expects this interrupt.

8) Trace Interrupt: An interrupt for use in tracing program execution. At the time of this writing, this interrupt is unimplemented.

9) Time Limit Interrupt: Generated when a task uses more system time than the upper limit established by the system configuration. The standard version of UniFLEX supplied at the time of this writing, however, imposes no upper limit on time for tasks.

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

12) User-Defined Interrupt: This interrupt has no defined meaning to UniFLEX. It is merely an additional interrupt which a user program or set of programs may issue and catch for whatever purpose they wish.

On return from a "cpint" call, an address is returned in the 6809's D register. This address is the address which the system was previously using on receipt of program interrupts. In other words, it is the address which was provided in the previous "cpint" call. This old address can be used to tell what kind of action a program was taking on receipt of program interrupts before the current "cpint" call. For example, assume we have a program that is ignoring quit interrupts. If we now issue the instruction:

```
sys cpint,QUITI,0
```

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

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

```
          ...
          ...
          sys   cpint,INTI,1       start by ignoring
          cmpd  #1                 was program A ignoring?
          beq   contin             if so, then so should we
          sys   cpint,INTI,handle  if not, catch it
contin    ...
          ...
```

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

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

```
          ...
          ...
          sys   cpint,ALRMI,wake  catch alarm & goto wake
          ldd   #30               delay 30 seconds
          sys   alarm
          sys   stop              wait for alarm interrupt
          ...                     continue with program
          ...
          ...
wake      rti                     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". In this example the code at "wake" does absolutely nothing but return. That is because when the alarm is received we want to simply continue execution of the program where we left off 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 then see any waiting program interrupt. There are a few certain calls, however, which may be terminated by a program interrupt. In particular, those system calls which may be interrupted are "read" and "write" (if the device being read or written is a slow device such as a terminal or printer) and the "stop" and "wait" calls. A read or write call to a fast device, such as a disk file, will never be terminated by a program interrupt. If a program interrupt does get through to one of the system calls mentioned above, the following action takes place. First the system call is immediately terminated and control is passed to the program interrupt handling code if the interrupt is being caught. Then, when the interrupt handling code is complete, control is passed to the instruction immediately following the interrupted system call AND AN ERROR STATUS IS RETURNED. This error status is accompanied by an "EINTR" error (number 27). In this way, the program which made the system call can detect that it was interrupted and re-issue the system call if desired.

As an example, consider a program which prompts the user for a line of data from the terminal. If a program interrupt is sent to that program while in the middle of the read system call to get the data from the terminal, that call may be prematurely terminated; ie. 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. An alternative, however, would be to recognize that it was an EINTR error and loop back in our code to re-issue the prompt and the read system call to input the data again.

6.5 Locking and Unlocking Records

In a multi-user, multi-tasking system such as UniFLEX, it is possible to have more than one user or task attempting to access the same file. In some instances this is no cause for concern and indeed is often beneficial. There are times, however, when it could be disastrous. For example, consider an airline reservation system. Assume one user reads in a record containing available seating information about a flight and finds there is only one seat left. Normally he would type in the necessary information about the passenger and when complete, the system would rewrite that record to show there were no longer any seats 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 would be able to access it. In our airline example above, the first user would lock the record just before reading it and not unlock it until he was finished writing the new data out. When the second user attempted to read the record, he would see it was locked and would have to wait until it was unlocked.

UniFLEX maintains a table showing what records are locked in the system. These records may be of any length as specified by the task which performs the lock. Note that a single task may only lock one record in a file. However, it is possible for other tasks to have other records locked in that same file. It is also possible for a single task to have a record locked in more than one file at a time. Two system calls provide this record locking ability, "lrec" to lock a record and "urec" to unlock 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 if the calling task already has a record locked in this file. If so, any such record is unlocked before the new record lock can be made. Next the system checks to see if the record to be locked is available or if some other task may have previously locked some portion of it. If available for locking, the system makes an entry in the locked record table and returns to the calling task. If the desired record overlaps some portion of an already locked record, the system returns with an ELOCK error. At this point the calling program could take some appropriate action.

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

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 take upon themselves the responsibility of avoiding reading or writing to a locked record. This may be accomplished by attempting to lock records before reading or writing them. If the record is available, no error is returned and we can go ahead with the read or write. If an error is returned (ELOCK error), we know that someone else already has the record locked and we should take some other action. One possibility is to put our task to sleep for a few seconds and then try locking the record again. This should be done with the "alarm" and "stop" system calls (consult the Introduction to UniFLEX System Calls). Proper use of the lock and unlock calls will yield the same result as if locking actually did prevent another task from reading or writing. Note that locking and unlocking will not be necessary in all cases, only in those where a data file is shared and conflicts can occur.

An example of record locking and unlocking in use can be found in the UniFLEX BASIC. The BASIC interpreter itself always performs record locking and unlocking in any record I/O or virtual array accesses (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 unlocks it by getting a different record, explicitly unlocks the record, or closes the file. Before any random file access, 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 on to the user as BASIC error number 49.

6.6 Shared Text Programs

In a multi-user environment, there may be many instances where several users are running the same program simultaneously. This is very wasteful of system resources if each user must have a copy of that program in memory at the same time. It would make good sense to let all the users share one copy of the program. This could only work if nothing was ever altered in the memory which contained that program. Now there is seldom such a program, but an alternative would be to separate a program into two portions, one which had code that would never be changed and one which had temporary storage and data that might require changes. Users could then share the non-changing portion and have their own individual memory assigned to each for the changing portion. That is the exact technique employed by UniFLEX which we call "shared text".

In UniFLEX it is possible to separate an assembly language program into two sections, a "text" segment for non-changing memory or memory which will only be read and a "data" segment for memory which can be changed by writing into it. When a user runs this program, both segments will be loaded into memory being separated by a memory page boundary. If a

second user runs the program at the same time, the system will recognize the fact that it already has a copy of the text segment in memory and will only load 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 when he runs.

The UniFLEX 6809 Assembler has the ability to generate programs that utilize this shared text feature as well as those which do not. If a program only uses 100 bytes for the text segment and 100 bytes for the data segment, it would not be beneficial to run as shared text. This is because the text segment must be in a different memory page than the data segment, thus requiring two pages or 8K to run our 200 byte program. In such a case the program should probably be assembled as absolute code. Generally speaking, a program should have a text segment of at least 8K before considering making it a shared text program. Shared text programs find their best use when the program is an interactive program such as an editor. This is because such a program usually runs for a relatively long period of time and there is a much better chance of more than one person using it at the same time.

For more details on how to assemble a shared text type program, refer to the UniFLEX 6809 Assembler manual.

7.0 General Programming Practices

There are several general programming practices which should always be followed when writing assembly language programs to run under 6809 UniFLEX. This section covers those points.

1) Starting Location

By default UniFLEX assumes all assembly language programs load and begin execution at location $0000. It is recommended that all user-written programs follow this convention by being assembled at $0000 and having an execution transfer address of $0000.

2) Stack Considerations

When a program begins execution under UniFLEX it is assigned a portion of memory to contain the program stack. The 6809 cpu's system stack pointer is left pointing somewhere within this memory. The user's program should NEVER write into locations in memory higher than this initial stack pointer location. The passed parameters which lie directly above the stack pointer (higher in memory) may certainly be read, but nothing should be written above the initial stack pointer location.

3) 6809 Interrupts

In general, a user program should not attempt to perform any 6809 interrupt-related code with the exception of the "SWI" interrupt instruction and system calls which generate "SWI3" instructions. Certain versions or configurations do permit user control of the "SWI2" interrupt via the "trap" system call, but a user should be certain he has one of those configurations before attempting to include any "SWI2" instructions in his code. More importantly, a user program should NEVER set the IRQ interrupt mask. Most hardware configurations do not support any means of protecting against such an action and the system can be "locked up" indefinitely.

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. Because of the multi-user environment and task switching, a delay loop does not provide accurate timing delays anyway. The preferred method is to use the alarm system call followed by a stop system call. The program must also then use the cpint system call to catch the alarm interrupt and continue with the desired code.

5) System "lib" Files Provided

Several system library files are provided on the master UniFLEX disk for the convenience of the assembly language programmer. Located in the "/lib" directory, these files contain definitions for several system related calls, tables, buffers, etc. The programmer may include these definitions in his programs by simply using the "lib" instruction in the 6809 assembler. These files are as follows:

sysdef	UniFLEX system call definitions
syserrors	UniFLEX system error definitions
sysints	Program interrupt definitions
sysstat	Status and ofstat buffer layout
systim	Time and ttime buffer layouts
systty	Ttyget and ttyset buffer layout

6) Generating Unique Filenames

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

8.0 Debugging

The assembly language debugging facilities provided as a part of UniFLEX are at best crude. In this manual we must assume that the system programmer does not have any sort of debugging program since one is not included in the standard set of supplied UniFLEX commands. In fact, the only tools provided for assembly language program debugging are the software interrupt, core dump, and "qdb" command. The 6809's software interrupt instruction (swi) can be assembled into the program to act much like a breakpoint. When the software interrupt instruction is hit, a core dump will be generated and the program will be immediately terminated. The system programmer may then examine the core dump with the qdb utility to determine the state of the memory, stack, and registers when the software interrupt was hit. The reader is referred to the UniFLEX Utility Commands manual for directions on the use of qdb.

The "qdb" utility will not display memory in the text portion of a shared text program. The data and bss sections are the only ones displayed. The user will find that the addresses which qdb uses for displaying memory from such a data segment are relative addresses from the beginning of that segment and not absolute addresses. Thus if you are examining a core dump from a shared text program whose data segment starts at $2000, telling qdb to display all memory from $0100 to $03B0 would display the data that was in absolute memory locations $2100 through $23B0. Note that data segments in a shared text program always start on a memory page boundary (4K boundary).

Most 6809 UniFLEX hardware systems do not provide a means to protect the system or other users from a "runaway" assembly language program. For this reason, testing of unknown assembler programs is very dangerous with multiple users on the system. It is highly recommended that assembly language programs be tested only when no other users are on-line, no other tasks are active, and all important files on any disk device which is on-line are backed up.

In the same vein, most 6809 UniFLEX hardware does not prevent the user from setting the IRQ interrupt mask. If this mask is set by a running assembler program, the system will be locked up with no means of recovery. It is imperative that no assembly language program sets the IRQ mask!

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 is a somewhat useful utility which reads a file (or list of files) and strips out all control characters except for carriage returns ($0d) and horizontal tabs ($09). We will name the utility "strip". The syntax of the command line is as follows:

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

The square brackets indicate that the file name specification is optional. If no filename is supplied (ie. the command line is simply "strip"), strip will read the standard input. The three periods ("...") indicate that it is possible to supply more than one file name. In such a case, strip will read all the files in order and write the stripped output to the standard output.

Our basic task, then, is to read either a list of files or the standard input, strip the necessary control characters, and write the result to the standard output device. In order to handle any size file(s), we shall read and write the data, buffer at a time. The question arises as to what buffer size 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 such that the program and the buffer could easily fit in one 4K memory page. Since this utility will probably not be frequently used, it was decided to limit the program memory utilization to only one page or 4K. We will make the read/write buffer as large as possible within that 4K space and still be a multiple of 512 bytes. The technique to accomplish this will be pointed out later.

The printed listing of the "strip" utility follows shortly. It is heavily commented 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 on line 17. Next we actually begin our program at location 0000 with the "org" statement in line 22. In line 26 we load the "U" register with a pointer to the list of filename arguments (the list is null if there was no filename specified). Notice we skip four bytes, two containing the argument count and two containing argument 0 which is the name of the command itself. Lines 27 through 32 check to see if a file or files were specified on the command line. If so, the argument count (at 0,s) will be greater than one because argument 0 (the command name) counts as one. If the argument count is equal to one, no file was specified so we must read the standard input. The file descriptor for standard input is 0 so that value is saved in "ifd" and we jump ahead to process that input. If a file was specified we enter a loop to read through all specified files. In line 36 we obtain the pointer to the next file in the list.

If that pointer is zero (a null pointer), we have reached the end of the list and we jump off to the exit code at "done". If it is non-zero, it must be the address of a filename string. Lines 41 through 44 open that file for read and save the file descriptor in "ifd". Note that the open is done via an indirect system call. This is necessary because when the program is written we do not know what filename to specify in an open call. The pointer to the name of the file to be opened is only discovered as we run the program. Notice on line 41 that this filename pointer is stored in the parameter list for the upcoming indirect open system call. In line 48 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 49 branches back to the top of the loop to look for another possible input file.

The "strip" subroutine is where the actual control character stripping work is done. In lines 64 through 66 we read "BUFSIZ" characters into memory at "buffer". Lines 70 and 71 check for end of file. If we were at the end of the file, we jump to "strip9" and exit the subroutine. If not, we go on to lines 77 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 the control characters are stripped, the resulting data is left in the same buffer. Because some characters may have been stripped out, the location of the end of the data in the buffer may be lower than before the stripping. After the stripping, we fall into lines 96 through 102 where the stripped data is written out to standard output. Lines 96 through 98 calculate the number of characters to write. It is equal to the difference between the pointer to the end of the data in the buffer and the pointer to the beginning of the buffer. The result is stored in the parameters for an indirect write call. In line 99 we obtain the file descriptor for the standard output file. Then the indirect write system call is carried out in lines 100 and 101. In 102 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. Upon receipt of an error, we simply write an appropriate message to the standard error output (file descriptor 2). The important thing to note about this code is that we save the error status so that it may be passed on to the "term" system call.

Lines 141 through 152 contain temporary storage and buffers. First are the parameter lists for the indirect open and write calls mentioned earlier. Line 150 reserves storage space for the current input file descriptor. Lines 152 through 154 reserve the read/write buffer. As explained above, we decided to make the buffer as large a multiple of 512 bytes as possible and still fit within 4K. 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 4K memory page. The org statement on line 152 ensures that we start on a 512 byte boundary. Recall that read/write efficiency is gained not only by a buffer size which is a multiple of 512 bytes, but also by beginning the buffer on a 512 byte boundary. Line 154 establishes the buffer size by calculating the difference between the end of the first 4K page ($1000) and the beginning of the buffer.

The "end" statement on line 157 has the utility starting address specified in its operand field.

There is no better way to learn programming than by doing, so it is highly recommended that as a starting point the reader type in, assemble, and execute this utility.

```
1=  ************************************************************
2=  *
3=  * UniFLEX "strip" Utility
4=  *
5=  * Copyright (c) 1981 by
6=  * Technical Systems Consultants, Inc.
7=  *
8=  * Utility to strip all meaningless control characters from
9=  * input file and write stripped version to standard output.
10=  * Accepts list of input files or defaults to standard input.
11=  * For the purpose of this utility, "meaningless control
12=  * characters" are all characters with an ASCII value between
13=  * $00 and $1f inclusive except carriage return ($0d) and
14=  * horizontal tab ($09).
15=  ************************************************************
16=
17=          lib     sysdef      read system definitions
18=
19=  ****************************************
20=  * start of main program
21=  ****************************************
22=          org     0
23=
24=  * start by seeing if any input files were specified
25=
26=  start   leau    4,s         set arg ptr past count & arg0
27=          ldd     0,s         check argument count
28=          cmpd    #1          file(s) specified only if >1
29=          bhi     main2       branch if filenames present
30=          ldd     #0          else use standard input
31=          std     ifd         save input file descriptor
32=          bra     main4       go process std. input
33=
34=  * check to see if any more files specified
35=
36=  main2   ldx     0,u++       get next argument in list
37=          beq     done        branch if no more args
38=
39=  * open specified file for read
40=
41=          stx     opname      put filename in parameters
42=          sys     ind,iopen   do indirect open call
43=          bes     opnerr      branch if error
44=          std     ifd         save input file descriptor
45=
46=  * strip control characters from this file
47=
48=  main4   bsr     strip       subroutine to strip CTRLs
49=          bra     main2       look for more files
50=
51=  * finished all input files, terminate task
52=
53=  done    ldd     #0          show normal termination
54=          sys     term
```

```
55=
56=   ****************************************
57=
58=   * subroutine to strip meaningless control characters
59=   * from the file specified by file descriptor in "ifd"
60=   * and write result to standard output.
61=
62=   * begin by reading a buffer full
63=
64=   strip   ldd    ifd         get input file descriptor
65=           sys    read,buffer,BUFSIZ read buffer full
66=           bes    rderr       branch if read error
67=
68=   * check for end of file (0 characters read)
69=
70=           cmpd   #0          end of input file?
71=           beq    strip9      exit if so
72=
73=   * do actual stripping of control characters.  this will
74=   * be done in place in the buffer by collapsing the data
75=   * as meaningless control characters are stripped.
76=
77=           pshs   u           save contents of u
78=           tfr    d,x         put character count in x
79=           ldu    #buffer     point u to buffer
80=           ldy    #buffer     point y to buffer
81=   strip4  lda    0,u+        get a character
82=           cmpa   #$1f        a control character?
83=           bhi    strip5      go keep character if not
84=           cmpa   #$0d        a carriage return?
85=           beq    strip5      keep if so
86=           cmpa   #$09        a tab?
87=           bne    strip6      if not, don't keep
88=   strip5  sta    0,y+        put char. back in buffer
89=   strip6  leax   -1,x        decrement count
90=           bne    strip4      loop if more characters
91=           puls   u           restore u register
92=
93=   * finished stripping, y points to end of buffer of
94=   * stripped data ready to be written
95=
96=           tfr    y,d         calculate no. of chars. to
97=           subd   #buffer     be written out
98=           std    wrtcnt      store in parameters
99=           ldd    #1          write to standard output
100=          sys    ind,iwrite do indirect write
101=          bes    wrterr      branch if error
102=          bra    strip       go read another section
103=
104=  strip9  rts                exit routine
105=
106=
107=
108=
```

```
109=   ****************************************
110=
111=   * error handling routines
112=
113=   opnerr  pshs    d           save error status
114=           ldd     #2          standard error output
115=           sys     write,opners,opnerl
116=           bra     err
117=   rderr   pshs    d           save error status
118=           ldd     #2          standard error output
119=           sys     write,rderrs,rderrl
120=           bra     err
121=   wrterr  pshs    d           save error status
122=           ldd     #2          standard error output
123=           sys     write,wrters,wrterl
124=
125=   err     puls    d           restore error status
126=           sys     term        exit program
127=
128=
129=   opners  fcc     "Can't open input file.",$d,0
130=   opnerl  equ     *-opners
131=   rderrs  fcc     'Error reading input file.',$d,0
132=   rderrl  equ     *-rderrs
133=   wrters  fcc     'Error writing output file.',$d,0
134=   wrterl  equ     *-wrters
135=
136=   ****************************************
137=
138=   * temporary storage and buffers
139=
140=   * indirect open system call parameters
141=   iopen   fcb     open        open function code
142=   opname  fdb     0           name of file to open
143=   opmode  fdb     0           open mode 1 (reading)
144=
145=   * indirect write system call parameters
146=   iwrite  fcb     write       write function code
147=   wrtbuf  fdb     buffer      buffer to write from
148=   wrtcnt  fdb     0           byte count to write
149=
150=   ifd     rmb     2           input file descriptor
151=
152=           org     (*+511)&!511
153=   buffer  equ     *           start on 512 byte boundary
154=   BUFSIZ  equ     $1000-buffer multiple of 512 bytes
155=           rmb     BUFSIZ      reserve space for buffer
156=
157=
158=           end     start
```

APPENDIX A
Alphabetical Summary of UniFLEX System Calls

call	no	description	syntax
alarm	43	sleep for some seconds	CALL: seconds in D sys alarm RTRN: previous seconds in D
break	6	exetend memory address	sys break,address
cdata	36	request contiguous data	sys cdata,address
chacc	25	check access permission	sys chacc,fname,perm
chdir	21	change directory	sys chdir,dirname
chown	23	change file owner	sys chown,fname,ownerid
chprm	24	change access perm	sys chprm,fname,perm
close	15	close file	CALL: file descriptor in D sys close
cpint	8	catch program interrupt	sys cpint,interrupt,address RTRN: old address in D
create	11	create a file	sys create,fname,perm RTRN: file descriptor in D
crpipe	31	create pipe	sys crpipe RTRN: read file descriptor in D write file descriptor in X
crtsd	20	make special file or directory file	sys crtsd,fname,desc,address
defacc	26	set default access	sys defacc,perm
dup	16	duplicate open file	CALL: file descriptor in D sys dup RTRN: new file descriptor in D
dups	17	duplicate specific file	CALL: current file descriptor in D requested file descriptor in X sys dups RTRN: new file descriptor in D
exec	2	exec	sys exec,fname,arglist

filtim	52	set file time	CALL: hi order time in X lo order time in D sys filtim,fname
fork	3	fork	sys fork RTRN: new task rtrns after call old task rtrns after call + 2 old task has new task id in D
gtid	32	get task id	sys gtid RTRN: task id in D
guid	33	get user id	sys guid RTRN: actual user id in D effective user id in X
ind	0	indirect call	sys ind,label
indx	1	index indirect call	sys indx
link	18	link to file	sys link,fname1,fname2
lock	22	lock task in memory	sys lock,flag
lrec	47	lock file record	CALL: file descriptor in D sys lrec,count
mount	29	mount device	sys mount,sname,fname,mode
ofstat	27	get open file status	CALL: file descriptor in D sys ofstat,buffer
open	10	open file	sys open,fname,mode RTRN: file descriptor in D
profil	37	profile task	sys profil,prpc,buffer,bsize,scale
read	12	read file	CALL: file descriptor in D sys read,buffer,count RTRN: bytes read in D
sacct	50	system accounting	sys sacct,fname
seek	14	seek to file position	CALL: file descriptor in D sys seek,positionhi,positionlo,type RTRN: hi position in X lo position in D
setpr	35	set priority bias	CALL: priority in D sys setpr
spint	9	send program interrupt	CALL: task number in D sys spint,interrupt

stack	7	grow stack	CALL: address in X sys stack
status	28	get file status	sys status,fname,buffer
stime	40	set time	CALL: hi order time in X lo order time in D sys stime
stop	44	stop until interrupt	sys stop
suid	34	set user id	CALL: user id in D sys suid
term	5	terminate task	CALL: status in D sys term
time	39	get time	sys time,tbuf
trap	38	set swi2 trap vector	sys trap,address RTRN: previous trap address in D
ttime	41	get task time	sys ttime,buffer
ttyget	45	get tty status	CALL: file descriptor in D sys ttyget,ttbuf
ttynum	51	get tty number	sys ttynum RTRN: tty number in D
ttyset	46	set tty status	CALL: file descriptor in D sys ttyset,ttbuf
unlink	19	unlink from file	sys unlink,fname
unmnt	30	unmount device	sys unmnt,sname
update	42	update file systems	sys update
urec	48	unlock file record	CALL: file descriptor in D sys urec
wait	4	wait	sys wait RTRN: task id in D termination status in X
write	13	write file	CALL: file descriptor in D sys write,buffer,count RTRN: byte count written in D

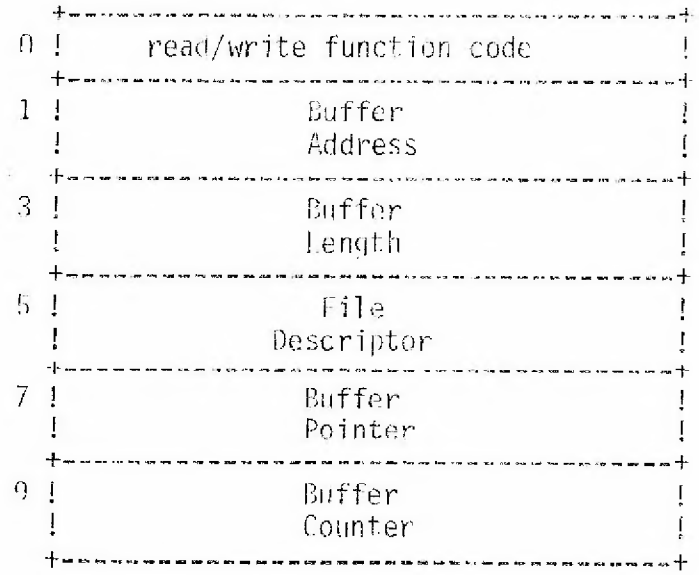
APPENDIX B
Sample Single Character I/O Routines

This appendix describes a set of primitive "character at a time" I/O routines which the UniFLEX assembly language programmer may use. These routines work for both terminals and files and relieve the programmer from the burden of managing I/O buffers. A listing of the routines follows this discussion.

I/O Block

Information about the file and its buffer is kept in an 11 byte array called an "I/O block". The I/O block contains the file descriptor, buffer pointers, and a read or write function call. The first 5 bytes of the I/O block are this read/write function block, consisting of the UniFLEX function code, the buffer address, and the buffer length. Following this is the file descriptor. The next four bytes have different meanings depending on whether the I/O block is being used to read or to write a file. If for reading, two bytes point to the next character to come out of the buffer and the last two bytes are a count of the number of characters remaining in the buffer. If this count of characters is zero, a "read" system call is issued to fill the buffer. If the I/O block is used for writing to a file, the first two bytes are the address of the next empty location in the buffer, and the next two bytes are a count of the number of characters in the buffer. If this count is equal to the size of the buffer, the buffer is written.

The following is a picture of an I/O block.



To assist in setting up an I/O block, a macro, "iobloc", is provided. The macro is called as follows:

```
label iobloc code,buffer,length,descriptor
```

The "code" argument must be either "read" or "write". The "buffer" argument is the address of the buffer associated with the file and "length" is the size of the buffer. The "descriptor" is the file descriptor, if known. This will be known if the I/O block is associated with the user's terminal. If it is not known, a zero should be supplied and the proper value stored into the I/O block when the file is opened and the true descriptor is known.

gnc - get next character

The subroutine "gnc" is used to get the next character from the buffer. If there are no more characters in the buffer, a call is made to UniFLEX to read more data from the file or terminal. When the routine is called, the x-register must contain the address of the I/O block being used. On exit, the character will be in the a-register. If an error was detected when reading data from the file, the carry bit will be set. If an end of file was detected, the overflow bit will be set. A sample call would look like this:

```
    .
    .
ldx #iob (x)=I/O block address
jsr gnc  get a character
bcs err  if an error occurred
bvs eof  if an end of file
    .
    .
```

pnc - put next character

The subroutine "pnc" stores a character in the buffer. On entry, the x-register should contain the address of the I/O block being used, and the a-register should contain the character that is to be stored in the buffer. If the buffer is full, UniFLEX is called to empty it. If an error occurs when the buffer is being written, the carry bit is set. A sample call would look like this:

```
    .
    .
ldx #iob (x)=I/O block address
lda chr  (a)=character to be written
jsr pnc  store character
bcs err  if an error occurred
    .
    .
```

fob - flush output buffer

The buffer used when writing to a file is actually written to the file only when it becomes full. To force the data in the buffer to be written to the file at other times, the "fob" subroutine is used. This should always be called before terminating the task to guarantee that all data has been written. If the buffer is empty, "fob" returns without doing anything. Another time that "fob" should be called is when a prompt is written to a terminal. If the prompt is written with "pnc", it is stored in the buffer but not actually sent to the terminal. Calling "fob" will force it to be written. On entry, the x-register should point to the I/O block begin used. If an error was detected while emptying the buffer, the carry bit is set. A sample call looks like this:

```
    .
    .
ldx #iob (x)=I/O block address
jsr fob  flush the buffer
bes err  if an error occurred
    .
    .
```

```
**    iobloc - macro to initialize i/o block
*
*     call:
*     iobloc mode,buffer,length,descriptor
*
*     where:
*      mode  ----- "read" or "write"
*      buffer ---- buffer address
*      length ---- buffer length
*      descriptor  file descriptor


iobloc  macro
        fcb     &1          read/write code
        fdb     &2          buffer address
        fdb     &3          buffer length
        fdb     &4          file descriptor
        fdb     &2          character address
        ifc     &1,'read'
        fdb     0           character counter
        else
        fdb     &3          available byte counter
        endif
        endm


**    I/O Block Offsets


IOB_FC  equ     0           function code
IOB_BA  equ     1           buffer address
IOB_BL  equ     3           buffer length
IOB_FD  equ     5           file descriptor
IOB_CA  equ     7           next character address
IOB_CC  equ     9           character counter

IOB_LN  equ     11          length of I/O block
```

```
**   gnc - get next character.
*
*    entry (x)=i/o block pointer.
*
*    exit  cs if error detected, and
*          (d)=error response
*
*          cc if no error, and
*          vs if end of file
*
*          cc, vc if no end of file, and
*          (a)=character
*
*    All registers not returning a response are preserved.

gnc      pshs   b,y
         ldd    IOB_CC,x  get remaining character count
         bne    gnc1      if characters left in buffer
gnc0     ldd    IOB_FD,x  (d)=file descriptor number
         sys    indx      reload buffer
         bes    gnc3      if error
         std    IOB_CC,x  save character count
         beq    gnc2      if end of file
         ldd    IOB_BA,x  reset character pointer
         std    IOB_CA,x
         ldd    IOB_CC,x  (d)=character count
gnc1     subd   #1        count character
         std    IOB_CC,x
         ldy    IOB_CA,x  get character
         lda    0,y+
         sty    IOB_CA,x  update character pointer
         clrb             clear carry, overflow
         puls   b,y,pc    return

gnc2     orcc   #2        set overflow indicating end of file
         puls   b,y,pc    return

gnc3     cmpd   #EINTR    check error
         beq    gnc0      if interrupted call, re-issue
         orcc   #1        set carry
         leas   1,s       remove (b)
         puls   y,pc      return (d)=error code, cs
```

```
**    pnc - put next character.
*
*     entry (a)=character
*           (x)=i/o block pointer
*
*     exit  cc if no error, and
*           (a)=character
*           cs if error, and
*           (d)=error code


pnc       pshs   a,b,y
          ldd    IOB_CC,x   get remaining room count
          bne    pnc1       if room left in buffer
          ldd    IOB_FD,x   get file descriptor number
          sys    indx       dump buffer
          bec    pnc0       if no error
          cmpd   #EINTR     check error
          bne    pnc3       if not interrupted call
pnc0      ldd    IOB_BA,x   update character pointer
          std    IOB_CA,x
          ldd    IOB_BL,x   (d)=new remaining size
pnc1      subd   #1         count character
          std    IOB_CC,x   update remaining count
          ldy    IOB_CA,x   store character
          puls   a
          sta    0,y+
          sty    IOB_CA,x   update character address
pnc2      puls   b,y,pc     return

pnc3      leas   2,s        remove character and (b)
          orcc   #1         set carry, error detected
          puls   y,pc       return, cs, (d)=error code
```

```
**   fob - flush output buffer.
*
*    entry (x)=i/o block pointer
*    exit cs if error detected


fob     pshs   a,b
        ldd    IOB_BL,x  save buffer size
        pshs   d
        subd   IOB_CC,x  determine number of characters
        beq    fob1      if empty
        std    IOB_BL,x  set character count to write
        ldd    IOB_FD,x  (d)=file descriptor number
        sys    indx      dump buffer
        bec    fob0      if no error
        cmpd   #EINTR    check error number
        bne    fob2      if not "interrupted system call"
fob0    ldd    IOB_BA,x  update character pointer
        std    IOB_CA,x
fob1    puls   d
        std    IOB_BL,x  restore buffer size
        std    IOB_CC,x  reset available space counter
        clra             clear errors
        puls   a,b,pc    return

fob2    pshs   d         save error response
        ldd    2,s       restore buffer size
        std    IOB_BL,x
        puls   d         restore error response
        leas   2,s       remove (a), and (b)
        orcc   #1        set carry, error encountered
        rts              return
```