The UniFLEX® Operating System

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.

The UniFLEX® Operating System

I. Introduction

This document provides an overview of the UniFLEX® Operating System (UniFLEX® Registered in U.S. Patent and Trademark Office). Several of the important system features are described including a look at the user interface, the file system, and the program environment.

II. The User Interface

After a user "logs in" to the system, a prompt will be displayed on the terminal, signifying that the system is ready to accept commands. A program called the shell program is responsible for issuing this prompt. The shell program is the primary interface between a user and the operating system. It collects and interprets the commands typed from the terminal and send the necessary information to the operating system so that it can perform the requested operation.

Each command in UniFLEX has a unique name, which is somewhat descriptive of the actions it performs. As an example, typing "date" will cause the command named "date" to be executed. This command will display the current date and time on the terminal, just as the name implies. In general, a command line has the following form:

<command_name> <arg_list>

where <command_name> is the name of the program (file) to be executed, and <arg_list> is a list of arguments that is collected by the shell program and passed to the program to be executed as an array of strings. Because the shell program collects the arguments, individual programs do not have to be concerned with parsing the command line. The shell program will look several places in the system for the command name specified, including the user's working directory. This allows a user to have a "local" command with the same name as a system command.

When a command is executing, the user will usually wait until it finishes, at which time the shell program issues a new prompt. It is possible to interrupt most commands by typing the interrupt character (a control-C on most systems). This character will cause premature termination of the command and immediate display of a new prompt. Another similar character is the quit character (a control-\ on most systems) which will do exactly the same thing as the interrupt character, but which will also create a "core dump" in the user's working directory. A core dump is an exact image of the running program's memory contents at the time the quit character was typed. The

operating system supports several utilities which allow a user to examine this core file, which includes the contents of the processor registers and the stack at time of termination. This feature is obviously a very handy debugging aid.

When a command is executed, it will initially have three files associated with it. These are called the standard I/O files. One "file" is the user's keyboard (standard input); one is the user's terminal (standard output); and the last one is also the user's terminal (standard error). Most commands which perform I/O operations work with the standard I/O channels.

As an example, the "list" command will list or display the contents of a file or group of files on the standard output device. Since the standard output is normally the terminal, the file's contents will be displayed on the terminal. The shell program can change the meaning of the standard output to some other file. This process is called output redirection and can be done as follows:

```
list file >outfile
```

This command line would invoke the "list" command and pass the string "file" to the "list" command as an argument. The string ">outfile" would not be passed to the command because the symbol '>' has a special meaning to the shell program. This character tells the shell program to redirect the standard output from the terminal to the file whose name follows. In this example, the output of the "list" command would go into the file named "outfile" instead of to the terminal. If this file did not previously exist it would be created, and if it did exist, it would be truncated to zero length before being used. The fact that the shell program takes care of this redirection of output means individual commands do not need special code to handle the situation.

Input may also be redirected. As an example, the text editor normally gets its input from the terminal. It is, however, possible to create a file of commands which may be sent to the editor as follows:

```
edit file <script
```

In this case, the file of commands is called "script", and the input is redirected by the shell program as informed by the '<' character. This method of I/O redirection is quite powerful. It should be noted that this convention, as well as most of the other conventions in the UniFLEX shell program, have been closely modeled after the UNIX™ shell program (™Unix is a trademark of AT&T Bell Laboratories).

The mechanisms involved in the standard I/O scheme can be used to an even greater extent with the implementation of "filters". A filter is a program which takes some input data, manipulates the data in some way, and outputs the result. If a program reads the standard input for its data and outputs its results to the standard output, it can be used in a very powerful way. In particular, the output of one command may be used as input for another command. As an example:

```
sort test-data ^ reject ^ spr
```

This command line consists of three commands, "sort" with the argument "test-data", "reject", and "spr". The '^' character is another special character detected by the shell program. This separator causes any standard output generated by the command to its left to be sent as standard input to the command on its right. In this example, the sort command will sort the file "test-data" and send the sorted output to standard output. Since the shell program has set this output to go to the standard input of the next command, "reject" will operate on these data. The "reject" program reads standard input, removes all adjacent duplicate lines, and sends its output to standard output. Again, the shell program will send this output to the next command, "spr", which is a printer spooler. The spooler will take its standard input and print it on the printer. Note that all three commands are essentially run simultaneously. Any data output by the first command is immediately available to the second. This example shows how you can take three totally independent programs and make them work efficiently together. The mechanism used to connect these filters is called a pipe and is another feature in UniFLEX which has been modeled after UNIX. There are many filter programs in UniFLEX. Their power should be obvious.

The shell program can understand more than one command at a time. As an example:

```
dir; list rugs; date
```

The ';' character is used as a command separator and instructs the shell program to continue parsing the command line after the specified command has finished executing. In this example, the commands "dir", "list", and "date" would be executed in a sequential fashion.

It is also possible to have the shell program execute multiple commands simultaneously, or in the "background". The '&' character used as a command terminator (or separator) will cause the shell program to execute the specified command and immediately issue another prompt. As an example:

```
rel68k testit >output &
```

will invoke the assembler ("rel68k") on the file "testit" and redirect the output to the file "output". Since the command is terminated with an '&', the shell program will run the assembly in the background and not wait for it to finish before issuing the prompt. When the prompt appears, the user may run another command even though the assembly is not complete. The shell program will report a task identifier number (task ID) for all background commands executed. This identifier may be used to terminate the task if desired. Since the '&' may also be used as a command separator, the following is also valid:

```
rel68k file1 >out1 & rel68k file2 >out2 &
```

This line will cause two assemblies to be run in the background, one on "file1" and one on "file2". These assemblies could have been run sequentially in the background with their output sent to the same file with this command:

```
(rel68k file1; rel68k file2) >output &
```

The parentheses act like those in expressions, grouping parts of the command line which belong together. This same line without the parentheses would have run the assembler on "file1", sending its output to the terminal. When it finished, a new assembly would be run in the background on "file2", with its output redirected into the file named "output".

As mentioned previously, the shell program performs all of the command line parsing and simply passes the collected arguments to the executed command as an array of strings. Command line arguments may contain special pattern-matching characters recognized by the shell program. There are several forms of these matching characters. One is the asterisk, '*', which will match anything. Another is the question mark, '?', which will match any single character. Finally, the construct "[x-y]" will match any character or range of characters contained in the brackets. Several examples will demonstrate this feature.

```
list text*
rel68k source?.a
list *test[a-dr]
```

The first line will list all files which start with "text" and have anything following. The second line will assemble the files which start with "source", have any character next, followed by ".a". The last example will list all files which end with "test" followed by one of the characters 'a' through 'd' or the letter 'r'. The shell program not only does the matching, but also alphabetically sorts the resulting list of arguments.

Since the shell program is no different from any other program, it may also be executed as a command. An application of this is command files. A command file is nothing more than a file containing a list of commands, exactly as they would be entered to the shell program. As an example, suppose the two commands "date" and "dir" were executed one after the other frequently. A file could be created which contained the following lines:

```
date
dir
```

Assume this file has the name "dd". This file can be passed as input to the shell program with the following command line:

```
shell <dd
```

Because the shell program reads standard input for commands (which is normally the terminal), the input may be redirected to a file. In this example the shell program will read the file, execute the commands, "date" and "dir", and terminate. This example is not useful, but suggests how complex command files may be constructed and executed. It is actually possible to directly execute a command file without having to type "shell <", but this method will not be described here.

The shell program has many more features. Since it is the primary interface between the user and the system, it is important that it be powerful and easy to use. The UniFLEX shell program is both, and it will undoubtedly gain additional features in the future.

III. The File System

The UniFLEX operating system has three main functions: file maintenance, I/O control, and task scheduling. The structure of the file system is probably the most important, since design flaws here will impair almost every program run on the system. Here again, the system was modeled after the UNIX system.

The operating system supports three basic types of file: ordinary, directory, and special. The majority of files are ordinary files. These files are simply a collection of bytes without any special meaning. There is no concept of records and no forced structuring of data. All files may be accessed either sequentially or randomly and may be as large as one billion bytes.

Each file in the system is protected by a set of permission bits. These permission bits determine whether or not a file may be read, written, or executed. Two bits exist for each of these modes. One defines the permission for the file's owner; the other, for all other users. As an example, the owner of a file may set the permissions such that she or he may read or write the file, but all others may only read it.

The second file type is the directory. A directory is a specially constructed file that contains the names and identifies the locations of other files. The directories on the system form a hierarchical tree structure. The root of the tree is called the "root" directory. Any directory may contain entries which are names of other directories (or subdirectories). Each user of the system is assigned a directory. When a user "logs in", the assigned directory becomes the working directory. There is no limit to the number of directories on the system.

Since many files and directories exist on the system, a mechanism is needed for specifying a particular file in a specific directory. This mechanism is known as a file specification (or path name). A file specification is a description of the most direct path from the root directory through the directory tree to the file in question. A file specification is independent of the user's location in the directory tree because it always starts in the root directory. Because the slash character, '/', is used to represent the root directory, a file specification always begins with a slash. In addition, slash characters are used to separate the components of a file specification. For example, the file specification "/usr/john/test" tells the system to start in the root directory (specified by the leading '/' in the file specification), find the directory named "usr" in the root, then scan that directory for the directory named "john", and finally scan the directory "john" for the file named "test".

A path describing the whereabouts of a file that does not begin in the root directory is a file name, not a file specification. The system always begins such a search in the user's working directory, which can be anywhere in the directory tree. A file name describes the path from the user's working directory to the file in question. Thus, the same file has different names depending on the working directory.

The last component of a file specification is sometimes called the "simple file-name" or, more loosely, just the "file name". A simple file-name may not contain more than fifty-five characters. If the user specifies a longer name, the operating system truncates it.

Each entry in a directory may use up to 64 bytes. The operating system deals with these bytes in groups of 16. Each group of 16 bytes that is not currently in use consists entirely of null bytes. The first two bytes of each group of 16 are reserved for the number of the file descriptor node (fdn), if appropriate. The fdn is simply a 16-bit number used to identify the file on the disk. These first two bytes are used only in the first group of 16 bytes in any entry. If more than one group of 16 bytes is used in an entry, the first two bytes of the additional groups remain null bytes.

The third through sixteenth bytes in each group are used as necessary for the characters of the simple file-name. With one exception, the last character in a directory entry must be a null byte, which signifies to the operating system the end of the entry. Only if the simple file-name is exactly 14 characters long is there no such null byte. This behavior is necessary to maintain compatibility with the 6809 operating system. If the simple file-name is more than 14 characters long, the operating system turns on the upper bit of the third byte in the first group (the first character of the file name) to indicate that the line is incomplete. This behavior allows the system to distinguish between a file name that contains exactly fourteen characters (no bit set) and one that contains more than fourteen characters without having to look at the next line in the directory.

If the simple file-name contains more than 14 characters, the operating system must use more than one group of 16 bytes. When it adds the second or a subsequent group of bytes to an entry, it turns on the upper bit in the third byte of the new group (remember the first two bytes in all groups beyond the first remain null bytes) to indicate that the line is a continuation of an entry. It then uses as many bytes of that group as necessary for the name of the file and the null character that must terminate the name. Because the operating system truncates any name longer than 55 characters, the last character in the fourth group of bytes is always automatically a null byte.

All directories have at least two entries, one named ".", and one named "..". These names are purely convention. The file "." represents the directory itself, and the file name ".." represents this directory's parent directory. The "." entry is useful in referencing the working directory without knowing its name, and the ".." entry is used for reverse traversal of the directory tree.

The permission bits previously described also apply to directories. If the user who owns a directory read protects it, other users will not be able to display the contents of that directory. If the directory is write protected, no new files may be placed in the directory. If a directory is execute protected, it may not be searched for a specified file name or used as part of a file specification.

As an extension to the directory tree structure of a file system, another file system (disk unit or units) may be mounted at any node of the tree. The mounting process effectively replaces an existing node (directory) with the root directory of the mounted file system. As an example, a system with two disk drives will use one of the drives as the system root device, that is, the drive containing the directory known as '/' to the system. In order to access the directories and files on the second drive, it is only necessary to mount this device on an existing directory of the root device. The mounting operation will cause the contents of the selected directory to become inaccessible, replacing its contents with the root of the directory tree on the second drive. An unmount operation will restore the original directory. This procedure logically extends the notion of file names to allow access to any file on any currently mounted file system.

A specific example will clarify the mount operation. Let's assume there is a directory named "user2" in the root directory of the main system disk. Let's also assume that we have another disk which contains a file named "test" in a directory named "source" in the root directory of that disk. Performing a mount of this second disk onto the directory "user2" will now allow access of the file "test" with the following file specification:

```
/user2/source/test
```

Note that no mention of device name or device type was necessary to access this file. This structure allows several file systems to be connected together as one big tree, greatly simplifying overall file organization.

The third type of file in UniFLEX is the device (or special) file. All devices on the system appear as file names in directories, just as regular files do. All of the devices are normally kept in the directory "/dev". This means that programs which read and write file data may just as easily read and write data to and from a device. As an example, to write data to a printer, the program could write to the file "/dev/printer". Treating I/O devices in this way allows fairly device independent I/O, in that file and device I/O operations are very similar. It also allows the same protection scheme used for files to work for devices. This mechanism of device files, or special files is identical to that used by the UNIX operating system.

Since files and I/O devices are so similar, the same system I/O calls may be used for both. The UniFLEX system calls to perform I/O allow files to be created, opened, read, written, and deleted. The following examples show the calls as procedure calls in the C language. The call to open a file looks like this:

```
open(pathnam, mode)
```

where "pathnam" is the address of a character string containing a path name to the file to open, and "mode" specifies whether the file should be opened for read, write, or update (both read and write). The "open" call returns a value called the file descriptor, which is used to identify the file for future I/O operations. The file descriptor is simply a number which the operating system associates with the open file.

The "open" call requires that the specified file already exist. To create a new file (or truncate an existing file to zero length), the "creat" system call is used:

```
creat(path, perms)
```

This call also returns a file descriptor. The argument "perms" specifies which permission bits should be associated with the file. Once the "creat" has been executed, the file is left open for write.

To read data from an open file, the system call "read" is used:

```
read(fildes, bufad, nbytes)
```

The argument "fildes" is the file descriptor for the open file from which to read data. The argument "bufad" is the address of a buffer in which the system will place the data from the file. The argument "nbytes" specifies the number of bytes wanted from the file. The corresponding write operation is similar:

```
write(fildes, bufad, nbytes)
```

In this case, the system writes "nbytes" bytes from the buffer at "bufad" to the file represented by the file descriptor. Both the "read" and the "write" calls, return a value which is the actual number of bytes read or written. When writing, the returned value should always be equal to the number of bytes requested, or an error has occurred. The value returned by a "read" call need not equal the number of bytes requested. A returned value of 0 represents the end-of-file condition.

Reading and writing may take place in any part of the file. Each open file has a file pointer associated with it. Reads and writes start at the current position of this pointer and advance the pointer by the number of bytes transferred. An open operation sets the file pointer to the beginning of the file. The "lseek" system call allows repositioning of the file pointer. It has the following form:

```
lseek(fildes, offset, type)
```

where the file descriptor selects the file, and the "offset" is a byte count representing the relative position from the file's beginning, end, or current position, determined by the value of "type". This call returns the actual value of the resulting file pointer (bytes from the file beginning). Seeking beyond the end of a file and reading results in an end-of-file condition, while writing simply extends the file to include the written data. When the operating system extends a file, it allocates just enough disk space to record the new data. For example, although performing a seek to byte 10,000 in a file which has length of 100 and writing one character will produce a file of logical length 10,000, the system allocates only two disk blocks to the file. Reading data from the file will yield null bytes where no disk space is actually present.

The disk I/O facilities of UniFLEX are quite efficient, allowing full processor overlap with disk I/O transfers. The system maintains a disk-block buffer cache in main memory which may contain copies of from eight to sixty-four of the most recently accessed disk blocks (the actual number is system-dependent and may be altered by the "tune" command). When a program requests data from a particular disk block, the system either searches the buffer cache for that block or copies the data directly into the space allocated to the user, depending on which method is more efficient at the time.

UniFLEX also supports full read-ahead and write-behind data transfers. Read-ahead implies that whenever the system needs to read a block of a file, it will automatically read the next sequential block as well. Since the disk read operation is overlapped with the CPU operation, little, if any, time is wasted doing the additional read. Write-behind means that any information to be written to the disk is simply placed in one of the cache buffers and written at a convenient time. Programs writing data are not delayed until the write actually occurs. This combination of read-ahead, write-behind, and the buffer cache gives UniFLEX a superior I/O transfer rate.

UniFLEX also supplies a mechanism for locking records. This is one area where the UNIX operating system falls short. The following system call:

```
lrec(fildes, count)
```

will lock "count" bytes from the current file pointer in the file represented by the file descriptor. The count size or record size may be anywhere from 1 to 65535 inclusive. The locking action is more of a convention than an actual hard lock operation. After one program has locked a section of a file, other programs may still read or write that section of the file without error. If, however, another program tries to lock a section of a file which is already locked, an error will result. This structure has proven to be very efficient in that programs dealing with data-base files may make use of the lock mechanism and preserve the integrity of the data while those working with regular files need not be concerned. A locked record may be unlocked by another "lrec" call from the same program, by closing the file, or by issuing the "urec" call specifically to unlock the record.

There are several additional system calls in UniFLEX pertaining to I/O. These include calls to close, delete, and link a file as well as calls to create new directories, change a file's owner and permissions, and get a file's status.

IV. Task Structure

Each program under UniFLEX runs as a separate task. When a task is actively running, it has its own dedicated address space. This means that the task has the complete address space of the CPU and any part of this space will either contain memory or be totally void. No I/O devices or system code is present when the task is running. Each task is assigned enough memory to hold its program, data, and stack. The program (or text) size is set at the initial execution of the task and remains fixed. The data and stack segments may grow or shrink dynamically. The text part of a program may be shared among all tasks currently executing the same program. Sharing is accomplished automatically and tends to make more efficient use of available main memory. Optionally associated with each task is an "environment", which contains strings which may be meaningful to certain programs. The operating system keeps a large amount of information about each active task, including which user started the task, the task identifier, the current program size, amount of CPU time used, age of the task, and task

activity information. Tasks are scheduled CPU time based on their priority. The priority value is constantly adjusted by the system to reflect the current status.

New tasks are created by the "fork" system call. The fork call causes the calling task to duplicate itself, or split into two identical tasks. The complete address space of the calling task is duplicated for the new task, as well as the task's complete environment, including open files, and so forth. The new task starts execution upon return from the fork call. It may be distinguished from the parent in only one way. The fork call will return a value of 0 to the new child task, and a value which represents the child's task identifier (never 0) to the parent. This allows each task to determine if it is the child or the parent. The return from the "fork" is a little different at the assembly language level. Here, the return to the original task is two bytes beyond that of the new task. This allows the new task to perform a "branch" instruction before continuing. The child's task identifier is still returned to the parent task.

The operating system places no restrictions on what the new task can do. Normally, it will perform an "execl" system call which will invoke a new program. The form of the "execl" call is as follows:

```
execl(path, [arg0, [arg1, ...,[ argn,]]] nullp)
```

where "path" is the address of the character-string containing the path name of the file containing the program to execute. The calling task's address space is replaced by that of the called program. The "arguments" are made available to the new program as an array of strings. The first argument, "arg0", is by convention the name of the new program. Note that a return from an "execl" to the calling task is an error condition, usually because the specified file name was not found or not executable. The "execl" call can be thought of as a type of jump instruction where control passes to the first instruction of the called program. Most of the task's environment parameters, such as open files, are preserved across the "execl" call. Leaving files open permits the easy implementation of the standard I/O mechanism. All tasks usually start with the three standard I/O files already open. The file descriptors for these files are 0 for standard input, 1 for standard output, and 2 for standard error.

A task which "forks" another task may "wait" for the child task to terminate. The wait system call will block the calling task until one of its child tasks terminates. Upon termination, the "wait" call will return to the caller, returning the task identifier and the termination status of the dead task. Tasks normally terminate by the "exit" system call. It has the form:

```
exit(code)
```

where "code" is the termination code to return to the parent. A status of zero indicates normal termination, while a nonzero status specifies an error condition. A task may also be terminated by a program interrupt. Tasks have a choice of ignoring or catching these interrupts to avoid termination. As an example, the interrupt character (control-C) is sent as a program interrupt to all tasks associated with the terminal producing it. Normally, this will terminate the task, but programs like the UniFLEX Text Editor choose to catch this interrupt and take special action such as re-issuing the prompt to accept another command.

The operating system assigns a priority to each task. The task with the highest priority is always being run. A task's priority is constantly being adjusted to reflect its size, age, and CPU activity.

The system supports virtual memory for all tasks. When the system runs out of memory, it takes a 4-Kbyte page of memory from some task and uses that page (after writing its contents to the paging device if necessary). The page is restored from the paging device to the original owner when necessary. UniFLEX's scheduling routine is quite complex and tries to take in as many factors as possible when making scheduling decisions. As an example, tasks which have been ignored for a long time tend to increase in priority, and those which are monopolizing the system's resources are penalized. The idea is to be as fair as possible to all tasks in the system.

The amount of memory available to the task table is the only limit that the system imposes on the maximum number of tasks that can coexist. The number of tasks allowed may vary from 8 to 128. The system manager may adjust this number with the "tune" command. The restriction to a maximum of 128 tasks does not tend to be a serious restriction because hardware limitations are more likely to determine the useful maximum.

Several other system calls also pertain to tasks. These include calls to get a task's identifier, its owner, and one to incrementally adjust the priority over a small range. This last call is particularly useful for setting lower priorities for tasks which are typically background jobs.