The UniFLEX™ Operating System

Technical Systems Consultants, Inc.

The UniFLEX™ Operating System

COPYRIGHT INFORMATION

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 is a trademark of Technical Systems Consultants, Inc.). Several of the important system features are described including a look at the user interface, the file system, and the program environment. Several terms will be defined so that those not completely familiar with modern computer terminology may follow. At the end of this document, some hints of the system's efficiency are provided.

II. System Description

UniFLEX is the first full featured time-sharing system available for use on microcomputers. It provides both a multi-user and a multi-tasking computing environment. Its design was influenced by two previous operating systems, FLEX™ and UNIX™ (FLEX is a trademark of Technical Systems Consultants, Inc., and UNIX is a trademark of Bell Laboratories). FLEX has been a very popular single-user operating system. It has proven to be quite reliable and extremely flexible in a wide variety of applications. UNIX, on the other hand, has been one of the most popular multi-user operating systems ever created. To this date, it has only been available for minicomputers. Its size has not made it a good candidate for micros, but limited feature versions will probably be available for the upcoming 16 bit microprocessors. UniFLEX is the happy marriage of these two previous operating systems. It was designed to run with the "upper class" microcomputer, those which include memory mapping for large amounts of memory. No compatibility restraints were imposed during its design, hopefully eliminating inefficiencies and leaving old design flaws behind.

Many readers will not be familiar with large, multi-user operating systems. In the past, a person's exposure to a computer was typically through a time-sharing or large computer system. With the advent of the microcomputer, this is no longer true. A large percentage of individuals who deal with computers in their daily work activities have only been exposed to the single-user microcomputer system. With this in mind, the following discussion will point out some of the features needed in a multi-user operating system and how they apply to UniFLEX.

Multi-user implies that multiple users may make use of a computer's resources simultaneously. This implies that the operating system must be able to perform different operations or tasks simultaneously in order to serve its users' needs. The term multi-tasking describes this ability. Before a user may make use of the system, he must 'login'.

This process requires the user to enter a 'user name' and a password. The user name is used by the system to identify the person attempting to access the system, and the password is used to validate the user's authenticity. Most large scale operating systems (including UniFLEX) will keep a permanent record of the activity of each user. UniFLEX keeps a file called 'history' which notes the login and logout times of each user on a terminal by terminal basis. It also keeps information about system boots, date changes, and shut downs in this same file.

As each user attempts to run programs, the operating system must 'schedule' the CPU, allowing it to work on each job. It also must schedule the various I/O devices, such as disk drives, since it is very likely several programs will require disk data all at once. The scheduler is a program internal to UniFLEX which determines which task to run, and how long it should be permitted to run. In all cases, the task with the highest priority, which is ready to run, will be selected by the scheduler. A task's priority is determined by many factors but includes such things as memory size, age of task, ratio of I/O to compute time, etc.

Another function of the scheduler is 'swapping'. All large scale operating systems perform swapping but UniFLEX is the first to offer it on a micro. 'Swapping out' is the process of copying a task in memory to a secondary memory, such as a disk drive. 'Swapping in' is the reverse, copying the task from the disk back into main memory. The ability to swap tasks allows a system to run many more simultaneous tasks than can be kept in the system's memory. A multi-user or multi-tasking system without it is extremely limited.

As an example, let's consider a system which has 128K bytes of memory, of which, 32K is occupied by the operating system. This leaves 96K free for user programs. Now assume we have a small user environment of four active users on the system. Each user will run a different program, consisting of a text editor (which requires 28K for program and buffers), an assembler (44K), a BASIC interpreter (32K), and a Pascal compiler (40K). A system without swapping would only allow two of the users to start their programs. The other two would get error messages since there is insufficient memory to support their programs. These users would need to keep trying until one of the other user's programs finished. As you can see, this is not very practical.

A system which supports swapping would be able to run all of the programs in the 96K of available memory, even though the total program size is 144K. As above, two users would be able to start running their programs, but when the third and fourth programs are started, again there is not enough main memory. This time, however, the operating system will 'swap' the tasks out to secondary memory (a disk) and not issue the error. The scheduler will see that there are tasks swapped out which are ready to run and proceed to swap the tasks back and forth, effectively sharing the main memory among the various tasks. The swapping operation of course needs to be very efficient to avoid slow down in the system. UniFLEX, for example, can swap a typical task in about 100 milliseconds (approximately 180K per second transfer rate).

This time is using a Winchester type hard disk as a swapping device. The swapping time is 'overlapped' with CPU time which means no time is wasted for a task swap since the processor can be executing another task while a swap is taking place! With this swapping arrangement, there is essentially no memory imposed limit as to the number of active tasks in the system.

Large system operating systems typically offer many user oriented features. Some of these include 'mail', user communications, file protection schemes, and a 'startup' facility. UniFLEX offers all of these. The 'mail' program allows one user to send mail to any other user. This mail is simply a message which is tucked away in the receiver's files. Each time a user logs in to the system, he will automatically be informed of any mail received since the last login. This feature is extremely useful in an environment supporting many users. It is also possible to communicate directly with another user who is currently logged in to the system. A bidirectional communications path can be set up between the two users' terminals.

III. 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', is responsible for issuing this prompt. The shell is the primary interface between a user and the operating system. It will collect and interpret the commands typed from the terminal, and send the necessary information on to the operating system to perform the requested operation.

All commands in UniFLEX have a unique name and the names are somewhat descriptive of the actions performed by each. 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 argument1 argument2 ... argumentn
```

The 'command' is the name of the program (file) to be executed, and the 'arguments' are collected and passed on to this program as an array of strings. Notice that the shell does this argument collection, which means individual programs do not have to be concerned with command line parsing. The shell will look several places in the system for the command name specified, including the user's current directory. This allows a user to have 'local' commands with the same names as some system commands.

When a command is executing, the user will usually wait until it finishes, at which time a new prompt will be displayed by the shell. 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 backslash on most systems) which will do exactly the same thing as the

'interrupt' character, but also create a 'core dump' in the user's directory. A core dump is an exact image of the running program's memory contents at the time the quit character was typed. There are existing system utilities which allow a user to examine this core file as well as the processor registers and 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 and is called the standard error channel. 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 has the ability to 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' on to 'list' as an argument. The string '>outfile' would not be passed to the command because the symbol '>' has special meaning to the shell. This character tells the shell to redirect the standard output from the terminal, into the file name following. 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 takes care of this redirection of output means individual commands do not need special code to handle this situation.

Input may also be redirected. As an example, the text editor normally gets its input from the terminal. It is 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 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 have been closely modeled after the UNIX shell.

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 this 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. 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 the standard output. Since the shell has set this output to go to the standard input of the next command, 'reject' will operate on this data. Reject is a program which reads the standard input, removes all adjacent duplicate lines, and outputs through the standard output. Again the shell will send this output to the next command, 'spr', which is a system 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 very 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 type programs in UniFLEX and their power should be obvious.

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

```
asmb testit >output &
```

will invoke the assembler ('asmb') on the file 'testit' and redirect the output to the file 'output'. Since the command is terminated with a '&', the shell will run the assembly in the background and not wait for it to finish before issuing the prompt. At this time, additional commands may be run, even though the assembly has not completed. The shell will report a 'task identifier' number 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:

```
asmb file1 >out1 & asmb 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:

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

The parentheses act like those in expressions, and group 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 it output redirected into the file named 'output'.

As mentioned previously, the shell performs all of the command line parsing, and simply passes the collected arguments on to the executed command as an array of strings. Command line arguments may contain special pattern matching characters recognized by the shell. There are several forms of these matching characters. One is the '*' which matches 'anything'. Another is the '?' 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*
asmb 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 not only does the matching, but the resulting list of arguments which did match is sorted alphabetically.

Since the shell is no different than 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. 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 with the following command line:

```
shell <dd
```

Since shell reads the standard input for commands (which is normally the terminal), the input may be redirected to a file. In this example, the shell will read the file, and execute each command, date and dir. The shell will then terminate. This example is not very useful, but suggests how very 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 has many more features. Since it is the primary interface between the user and the system, it is very important that it be powerful and easy to use. The UniFLEX shell appears to be both, and will undoubtedly gain additional features in the future.

IV. 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 UNIX file system was modeled quite closely.

There are three basic types of files, ordinary, directory, and special. The majority of files are ordinary files. These files are simply a collection of bytes, having no 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.

All files in the system are '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 defining the permission for the file's owner, and another one for the permission of all others. As an example, the owner of a file may set the permissions such that he may read or write the file, but all others may only read it.

The second file type is the directory. A directory is exactly the same as an ordinary file with the exception that the data in the directory is operating system defined. Each directory entry requires 16 bytes, 14 of which are used to store the file name, and the remaining two are used for the 'File Descriptor Node', or fdn for short. The fdn is simply a 16 bit number used to identify the file on the disk. There is no limit to the number of directories.

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 his own directory. When a user 'logs-in', this becomes his 'current directory'. 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 'path name'. The path name is a list of directory names separated by slashes, all followed by the file name desired. As an example, the path name '/usr/john/test' tells the system to start in the root directory (specified by the leading '/' in the path name), 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'. If a path name is specified without the leading '/', the search will start in the current directory as opposed to the root directory.

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 current 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 a user 'read' protects his directory, others will not be able to display the contents of the directory, and 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 as part of a path name.

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 path name:

/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. 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 a general high level language form. The exact calling sequence is defined by the actual language in use. The call to open a file looks like this:

open(name, mode)

where 'name' is the path name of the file to be opened 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 file opened.

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

create(name, permissions)

This call also returns a file descriptor. The 'permissions' argument specifies which permission bits should be associated with the file. Once the create has been executed, the file is left 'opened for write'.

To read data from an open file, the system call 'read' is used.

read(file-desc, buffer, count)

The 'file-desc' is the file descriptor returned by the open call. The argument 'buffer' is a pointer to the space where the system will place the data from the file. The 'count' argument specifies the number of bytes wanted from the file. The corresponding 'write' operation is similar.

```
write(file-desc, buffer, count)
```

In this case, 'count' bytes are written from 'buffer' to the file represented by the file descriptor. In both the read and the write calls, a value is returned which is the actual number of bytes read or written. When writing, the returned value should always be equal to the requested 'count', or an error has occured. The value returned by read does not need to equal the 'count', and a returned value of zero 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 'seek' system call allows repositioning of the file pointer. It has the form:

```
seek(file-desc, 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 will result in an end of file condition, while writing will simply extend the file to include the written data. It should be noted that file extensions allocate just enough disk space to record the new data. As an example, 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, but only two disk blocks will be allocated 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 used to keep the most recently accessed disk blocks in main memory. When a program requests data from a particular disk block, the system first searches its memory buffer cache for the block. If it is found, no disk transfer need be made. If it is not found, the oldest block in the cache is given up, and its corresponding buffer is replaced by the contents of the requested block.

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, very little, if any, time is wasted doing the additional

read. Write behind means that any data 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 block buffer cache, gives UniFLEX a superior I/O transfer rate.

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

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

There are several additional system calls in UniFLEX pertaining to I/O. These include file closing, deletion, and linking. Other calls exist to create new directories, change a file's owner and permissions, and get a file's status.

V. 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 excution 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. This is done automatically and tends to make more efficient use of available main memory. 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, etc. 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 zero to the new child task, and a value which represents the child's task identifier (never zero) 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.

There are no restrictions placed on what the new task can do. Normally, it will perform an 'exec' system call which will invoke a new program. The form of the 'exec' call is as follows:

```
exec(file-name, argument1, argument2, ..., argumentn)
```

The 'file-name' is the name of the program to be loaded and run. 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. Note that a return from an 'exec' to the calling task is an error condition, usually because the specified file name was not found or not executable. The 'exec' call can be thought of as a 'jump' type instruction where control is passed to the first instruction of the called program. Most of the task's environment parameters, such as open files, are preserved across the exec. Leaving files open allows for easy implementation of the standard I/O mechanism. All tasks usually start with three files already open known as the standard I/O files, as previously described. These files have file descriptors 0 for the standard input, 1 for the standard output, and 2 for the standard error channel.

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 children tasks terminate. 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 'term' system call. It has the form:

```
term(status)
```

where status is a value made available to the parent task. A status of zero indicates normal termination, while nonzero 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 Text Editor choose to catch this interrupt and take special action

such as re-issuing the prompt to accept another command.

Tasks are run on a prioritized basis, the highest priority always being run. A task's priority is constantly being adjusted to reflect its size, age, and CPU activity. Tasks may also be swapped to secondary storage if the demand arises. The swap algorithm has built in hysteresis to avoid swapping out a task which has just been swapped in but not permitted to run. 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 hogging the system's resources are penalized. The idea here is to be as fair as possible to all tasks in the system. There is only one system imposed limit to the maximum number of tasks permitted in the system at any one time, the amount of memory available for the 'task table'. This does not tend to be a restriction since other hardware limitations tend to determine the useful maximum.

There are several other system calls which 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.

VI. UniFLEX Overview

UniFLEX is a very complete multi-user, multi-tasking operating system. It is intended to run with larger microcomputer systems and is not well suited for the small memory, small disk systems. The decision to require memory segment management (not bank switching) and efficient disk devices eliminated all compromises in the design. Small machines should have small operating systems while sophisticated hardware configurations deserve nothing but the most sophisticated operating systems. Trying to write an operating system which works equally well with limited hardware configurations almost always results in a less than optimal system.

One question which always arises when discussing multi-user operating systems is 'How many users'? This is a difficult question to answer because there are so many variables. UniFLEX can support any number of users, but the practical number ranges from two to about twenty on the 6809, and up to thirty-two on a 16 bit microprocessor. In most environments, more terminals may be connected than the upper practical user limit since not all terminals will be in use at any given time.

Many factors determine the maximum user count. These include such things as the amount of main memory, the processor clock speed, number of different hard disk drives, number of hard disk controllers, the hardware I/O structure, the efficiency of the memory management unit, and response times desired. The amount of main memory affects the amount of swapping the system will perform. If a separate high speed disk is used for swapping, less main memory is required. If one disk is being used for all system and user files, as well as swapping,

additional memory will speed up the system significantly. The speed of the swapping disk is also very important. Those running with a floppy disk drive for swapping will see a definite decrease in system performance.

Some applications are very terminal I/O bound. Word processing is one example. A system will generally be able to support more terminals for word processing than the same system could support for scientific or engineering applications. Business applications also tend to be very terminal I/O intensive. Keep in mind, that a terminal which is running a program waiting for input, has almost no impact on the system. Those environments which present this condition the majority of the time will be able to support many more users than those which are constantly running compute bound programs. Programs which generate a tremendous amount of output will degrade the system if the output is displayed at high baud rates. This degradation may be overcome by a 'front-end' I/O processor.

The final consideration in determining the number of users is the response time required. Response time is defined as the interval of time from the instance a keyboard entry is made, until the expected response is obtained. In many environments, the response time is not critical. Many educational systems, for example, would rather support more users at the cost of response time, since more users reduces the cost per student. All of these considerations are not peculiar to UniFLEX, but apply to any multi-user system, regardless of size.

The efficiency of an operating system can be partly determined by the amount of overhead required to perform a particular operation. UniFLEX was designed to keep system overhead at a minimum. Much of the current overhead is hardware imposed, but future systems promise to improve on this.

Since file activity is usually the biggest bottleneck in multi-user systems, the file system must be very efficient. UniFLEX is very efficient, not only in file storage overhead, but also in file transfers. The overhead involved in file storage is determined by the directory space, the file status information, and the file mapping information. In all, this is typically less than 8% overhead, a figure which is very respectable.

The disk transfer rate is where UniFLEX really shines. As a comparison, consider the test presented in 'The Bell System Technical Journal', July-August 1978, pages 1950-1951. This test compared three mini-computer operating systems by simply timing a disk file copy. The file was 480 blocks in length (245,760 bytes) and was copied on a system which was otherwise idle. This same test was run under UniFLEX, on a Southwest Technical Products S/09 6809 computer system. The main system disk was a Century Data Marksman, which is Winchester technology and holds approximately 17 megabytes of formatted data. The 6809 was only running at one megahertz. The results of the test were as follows:

system	seconds	msec./block
UniFLEX	27	28.1
UNIX	21	21.8
IAS	19	19.8

Both UNIX and IAS were running on DEC PDP 11/70's. It is no surprise that UniFLEX places last, but it is a surprise that it is only about 23% slower than UNIX on an 11/70! Increasing the speed of the processor to two megahertz should bring this value even closer (the total time would probably be reduced to about 24 seconds). This test does not prove much, if anything, but it is an interesting comparison.

This document is not intended to be a complete description of the UniFLEX operating system. Instead, it presents some of the system's highlights hopefully of interest to the reader.