General UniFLEX® Information

I. General Information

The UniFLEX® Operating System (®UniFLEX Registered in U.S. Patent and Trademark Office) is a state-of-the-art operating system for microcomputers. The system's design has been influenced primarily by two other operating systems, FLEX™ and UNIX™ (™FLEX is a trademark of Technical Systems Consultants; ™UNIX is a trademark of AT&T Bell Laboratories). UniFLEX retains the flexibility and ease of use of FLEX while incorporating some of the widely accepted structures of UNIX.

UniFLEX is a true multi-tasking, multi-user operating system. Each user communicates with the system through a terminal and may execute any of the available system programs. This implies that one user may be running the text editor while another is running BASIC while still another is running the C compiler. Not only may different users run different programs simultaneously, but one user may be running several programs at a time. For example, a compiler run on one file could be started, while making changes in another file with the editor. The system would inform the user when the compilation is completed.

Each user must 'log in' before being permitted to use the system. The log in process requires the user to enter a user name, followed by a password. The system checks the validity of the password, and if accepted, the system command language program (the 'shell') is run. Full system accounting is supported by UniFLEX which will keep track of user log-in and log-out times, as well as the user's utilization of system resources. This is ideal for an educational environment since individual student's system time can be closely accounted for.

UniFLEX is a very complete operating system and has no practical limitations built into the software; almost all of them are hardware imposed. It supports a hierarchical file system allowing file sizes up to one billion bytes and disk capacities of over eight billion bytes. All files are fully protected. A user may read, write, or execute protect files on an individual basis. All system I/O is device independent since I/O devices and files are treated in an identical fashion. Any combination of interrupt driven devices may be attached to the system. Any running task may initiate another task in an asynchronous manner. Inter-task communication is also supported. Task swapping may take place on those systems incorporating an appropriate swapping device. In real time applications, where swapping may be detrimental, it is possible to lock a task in main memory. Virtual memory versions of the operating system also exist.

The user's main connection with the system is through the command language. This is the vehicle for inputting commands to the system. It supports a wide variety of features including parameter passing and various types of name pattern matching. Input to the command language may be from the terminal or from any file (including other devices). This enables very complex command files to be created and run at will.

We will not claim that UniFLEX is the 'ultimate' operating system. It is impossible to create a program such as an operating system with which no one can find fault. We do feel, however, that UniFLEX does represent the state of the art in operating systems and incorporates as many 'large system' features as is practical without losing the friendliness of the smaller system.

II. Documentation Description

The UniFLEX documents contained in this volume cover all aspects of the operating system. A brief description of each follows.

A. Introduction to UniFLEX

This manual is oriented towards the beginning user of UniFLEX. It provides enough information for the novice to start using the operating system efficiently. Topics such as logging in and logging out are covered, as well as an introduction to a wide variety of system commands and features.

B. UniFLEX Utility Commands

All of the standard UniFLEX commands are described in detail. The command's syntax, use, and examples are all given. The commands are arranged alphabetically for easy reference.

C. UniFLEX Text Editor

The UniFLEX text editor is probably used more than any other program. This manual contains a tutorial on the editor as well as a complete description of all editor commands and features. The tutorial should be read by all system beginners.

D. UniFLEX 68xxx Assembler and Linking-Loader

Assembly language programmers will want to make use of this manual. It describes all of the features and capabilities of the 68xxx relocating assembler and linking-loader.

E. The UniFLEX Operating System

A general overview of UniFLEX is presented in this document. A discussion of the file system structure, task structure, and system features is included. This is recommended reading for those desiring more information on the details of the operating system.

F. Introduction to UniFLEX System Calls

This manual describes the system interface at the assembler level. System call structure, system errors, and system definition files are all described. All of the UniFLEX system calls are completely defined and appear in alphabetical order for easy reference.

G. UniFLEX Programmer's Guide

This is essential reading for those writing programs which need to interface with the system. Examples are given for most of the system calls. File manipulation, task control, and program interrupts are some of the topics discussed.

H. Miscellaneous UniFLEX Documents

This is a catch all and should be used for the variety of UniFLEX notes which will likely appear in the future. Those with maintenance will occasionally receive 'Product Bulletins' which may conveniently be filed in this section.

III. Acknowledgements

UniFLEX is a result of various motivations. The FLEX™ Operating System, a standard in the field for 6800 and 6809 microcomputers, was certainly an influence. Monitoring its many years of use has given us insight into the needs and wants of microcomputer users. The UNIX™ Operating System was a major influence on the design of UniFLEX. The structure of 'pipes', 'forks', and most of the other internal features of the operating system were modeled after those of UNIX. UNIX has been around for over a decade and has proved to be a solid design. UniFLEX does offer a few 'improvements' not found in UNIX systems. One of these, which is quite important in this day of complex data base management systems, is the file record 'lock' mechanism.

The UniFLEX Text Editor is a descendent of the FLEX editor which is a model of the editor, 'QED', in wide use at Purdue University. This editor has proved to be very 'usable' and continues to be the standard editor provided with the operating system.

Introduction to UniFLEX®

Introduction to UniFLEX®

UniFLEX® (®UniFLEX Registered in U.S. Patent and Trademark Office) is a very friendly and powerful operating system. Its applications include systems development, text processing, business applications, and educational use. This document is intended to be an introduction to the system, providing the reader with enough information to start using the system effectively. Many of the examples are presented in a form which may be exactly duplicated on the system; therefore, it is highly recommended that this manual be used in conjunction with one of the system computer terminals. The best way to learn is by doing, so feel free to experiment with the various topics presented here.

To get started, find a system connected terminal which no one is using and power it on, if it is not already. At the bottom of the screen (or page) there should be a message "Login:". UniFLEX is requesting your "user name" which should have been assigned to you by the system manager. Without a valid user name, you will not be able to obtain access to the system. In response to the "login" message, type your user name in all lowercase letters if possible. After the name, you must type the RETURN key. This tells the system that all of your name has been entered. If all goes well, the system will now respond with "Password:". When you were assigned your user name, you should also have received a password. This password is the key to your protection on the system, so guard it with your life. Type your password into the system at this time, followed by the usual RETURN. Notice that the characters of your password are not displayed on the terminal as you type them. This is to keep some passerby from discovering it. The system will now check the validity of your name and password, and if all is well, you will see the "message of the day" printed at your terminal, followed by the system prompt "++". The two plus signs signify that UniFLEX is ready to accept commands. If the system does not respond this way, you probably better seek out someone familiar with the system to get assistance.

After using the system you will eventually want to quit. Terminating an operating session is known as "logging out". To do this, simply type "log" and a RETURN in response to the prompt. The terminal should end up in the state you first found it, with the "Login:" message displayed. It is very important for you log out of the system before leaving the terminal. If you do not, someone may start using the terminal with your user name. They may damage information you have entered into the system or do other nasty things, so always log out!

Once you have successfully logged in, and the two plus sign prompt is displayed, try typing "date". Again, type the RETURN key after "date" to inform the system you have completely entered your request. The system should respond with something like this:

```
++ date
10:51:46 Tue July 8 1980
++
```

This demonstrates the execution of the "date" command. Notice that the name of the command gave you an idea ahead of time as to what might happen. This is true of most of the commands on the system. What would have happened if you would have mistyped "date". Try it! For example:

```
++ datf
Can´t execute ´datf´
++
```

Of course you may mistype a command name which ends up being another existing command name in the system. If this happens, more or less unpredictable results may occur.

It is possible to correct typing mistakes on an entered line before the RETURN is typed. The control-H character will act as a backspace. Typing one will delete the previous character typed, typing two will delete the previous two characters typed, etc. The control-H is entered by holding down the control key while typing the letter ´H´. Some keyboards may also have a "backspace" key to perform the same operation. A second special character for typing corrections is the line-cancel character defined as a control-X. Typing a line-cancel character will effectively delete all of the characters entered so far on the current line. The system´s "ttyset" command may be used to redefine these two characters but details will not be given here.

One other special character you should be familiar with is the control-C. If at any time you enter a command which you need to stop prior to its completion, type a control-C. This is known as the "interrupt character" and is quite useful. To show that the program stopped prematurely, the string "INTERRUPT!" will be displayed on your terminal. In some programs, such as the text editor, the interrupt character will stop what the program is doing but leave you in the program. If you wish to momentarily stop a program which is printing on the terminal, the "escape" key may be typed. Pressing "escape" once will stop the output; pressing it again will restart it. One last feature concerning typing at your terminal is "type-ahead". This allows you to type data at any time, even if the system is printing on your terminal at the same time. Since the characters you type are displayed as you enter them, they will be mixed in with the characters the system is printing, but the system will get them in the correct sequence. This feature means you may type in commands before the previous command has finished!

Much of the time spent on the system will be spent creating or modifying files. A file is simply a collection of information or data kept on the system and given a name for later reference. The "text editor" is the program which allows you to easily work with these files. There is a complete manual describing the use of the editor (The UniFLEX Text Editor) but for right now, we will use it simply to create some short files. Let´s start with a file named "myfile" by typing the command "edit myfile". The screen should look like this:

```
++ edit myfile
New file being created
    1.00=
```

The command name is "edit" and you are telling the system to edit the file "myfile". The system reports that you are creating a new file and prompts with line numbers to let you know which line you are entering. In this file, type your name on one line, then your address on the next. Since you typed two lines (each ending with the RETURN key) the system will now be prompting for line 3 (3.00=). As the first 2 characters on this line, type "#s" followed by the RETURN key. This tells the editor you are all finished. Your screen should now look something like this:

```
++ edit myfile
New file being created
    1.00=Your name here
    2.00=Your address here
    3.00=#s
++
```

Note that the two plus signs are back, signifying the system is ready to accept another command. Now that you have mastered this, try creating another file in exactly the same manner, but give this one the name "file2".

You have now created two files, "myfile" and "file2". To verify this, type the "dir" command. The results should be something like this:

```
++ dir
file2           myfile
++
```

The dir command tells the system to display the contents of your directory. A directory is simply a place in the system where a collection of files is kept. Notice that your two files are listed by name, and sorted in alphabetical order. Now type "dir +l" ("el", not one) which should produce something like:

```
++ dir +l
file2        1   1  rw-rw-  user    11:52 Jul 16 1980
myfile       1   1  rw-rw-  user    11:48 Jul 16 1980
++
```

The ´+´ is normally used to specify optional additional information to a command. This method of specifying options is very common in most of the UniFLEX commands. In this case, the "+l" informs the dir command to produce a long listing of your files. Notice that not only have the file names been displayed, but also the date it was last changed, the file owner´s name (in this example, "user"), a file size indicator (the number after the name), and the sequence "rw-rw-" which designates the permissions associated with each file. In this case, the owner of the file and all other users may read and write the file.

As time goes on, you will probably collect many files in your directory. It is often desirable to display or list the contents of a file to refresh your memory as to its contents. The "list" command will do this for you. Typing the command:

```
++ list myfile
```

will display the contents of the file you previously created using the editor. This same command may be used to list several files, one right after the other. As an example:

```
++ list myfile file2
```

will display the contents of both files as if they were one larger file. The "page" command is another command which will display the contents of files. Not only will each line be listed, but the lines will be grouped into pages, each page having a header which includes the file´s name, the date and time, and a page number. For example, the command

```
++ page myfile file2
```

will display your files in a page-formatted form. This command does not run the files together as the "list" command does, but instead, will start each file at the top of a new page.

It is often desirable to rename existing files. Let´s suppose you decide the name "file2" is not very descriptive. A better name might be "testfile". Instead of having to use the text editor and re-typing the entire file, you can use the rename command.

```
++ rename file2 testfile
```

The file that used to be called "file2" now has the new name. To verify this fact, execute the dir command again. You should see the following:

```
++ dir
myfile          testfile
```

You can also try to list "testfile" just to prove the file is unchanged. Another way to rename a file is to move it. This has exactly the same effect as renaming it but may be looked at a little differently. The "move" command will move a specified file to another file. Therefore, we could have done our file renaming this way:

```
++ move file2 testfile
```

You would have ended up with exactly the same results. It is also possible to make a copy, or exact duplicate of a file. The copy will of course have to have a different name, but the contents of the file will be the same. If you type the command:

```
++ copy myfile junk
```

Doing a "list" of junk will show that it is exactly the same as "myfile". Note that "myfile" still exists. Since it is not a good idea to keep a lot of unused files around in the system, the "kill" command should be used to delete them. For example, the command

```
++ kill junk
```

will delete the copy of the file which was just made. You are now back to your original two files, "myfile" and "testfile". It should be noted that the "kill" command will delete the files you specify without any prompting, so be sure you really want a file deleted before using this command. Once it is gone, there is no getting it back!

As you use the system, more and more of its features will become useful to you. One of the very convenient offerings is the mail service provided. This service allows you to send mail or messages to other users on the system. In order to receive mail, it is necessary to install a mailbox in which your mail will be placed. A mailbox is nothing more than a file which has the special name ".mail". To create this mail box file, type:

```
++ create .mail
```

The "create" command will create an empty file and give it the name specified. Once you have done this, anyone on the system may send you mail. Notice that the dir command will normally not display this file name. This will be explained shortly.

How do you know when you have received mail? Thats easy! Every time you log in, the system automatically checks to see if there is anything in your mailbox (.mail file). If it finds something, the message:

```
     You have mail.
```

will be printed on your terminal immediately following the message of the day. To view your mail, simply type the command "mail". Your messages will be displayed on your terminal and then you will be asked if you want your mail saved. Usually you will respond with an ´n´ for no and a RETURN, but other responses are possible. You should consult the complete description of the mail command in the manual <u>UniFLEX Utility Commands</u> for complete details.

Sending mail to someone else is also very easy. Let´s suppose the mail is to go to john, and "john" is his user name. Typing the command:

```
     ++ mail john
```

will allow you to type your message. No prompts will be issued until your message is complete. To signify the end of the message, type a control-D as the first character of a new line. The UniFLEX prompt should then be displayed. The control-D is a common character in the system and is used to signify "end of file". It tells the mail command that there is no more information. You will not be able to send mail to someone who has not installed a mailbox (created a file named ".mail") and the mail command will issue a message if this is the case. For practice, you can send mail to yourself (which is also a handy way to send yourself little reminders). This is not the only way to send mail. There are more details in the complete description of the "mail" command as mentioned earlier.

After creating your ".mail" file, you might notice that executing the "dir" command does not display that file name. This is a feature of the "dir" command, and is provided to avoid cluttering up your display with files you do not need constant reminding of. Normally, this command will not display any file name which starts with a period. If you use the "+a" option with "dir", you will see a list of the names of all the files in the directory:

```
     ++ dir +a
     .           ..          .mail        myfile       testfile
```

Notice that not only your ".mail" file is there, but also two additional files, "." and "..". These files are in every directory and are provided for convenience. The "." file represents the directory, and the ".." file represents its parent. More on these two files later.

The mail command is used to send a message to another user who is not necessarily on the system at the time. The "send" command may be used to communicate directly with another user who is currently logged in. To find out who is currently on the system, type the "who" command, which will display something like:

```
++ who
john    tty06    10:42 Jul 16 1980
jane    tty08    12:13 Jul 16 1980
user    tty09    12:17 Jul 16 1980
```

This will tell you the user names of those logged in, as well as the system's name for their terminal and the time at which they started. To send a message to user "john", type:

```
++ send john
```

Then type your message. If john were to send to you, the string:

```
Message from john
```

would appear on your terminal, followed by his message. It is possible to carry on a two-way communication if each user sends to the other. See the full description of this command for more details.

As you learn to use more commands on the system, it may be hard to remember the exact calling sequence of some of them. The "help" command is available to offer assistance. For example:

```
++ help who
```

will provide a short description of the "who" command and how to call it.

```
++ help list date
```

will give you information about the "list" and "date" commands. If you forget a command name you can type:

```
++ help
```

which will give you the names of most of the commands on the system. It will then prompt you for the name of a command with which you need help. Typing only a RETURN will exit the help command.

Back when the original two files were created, we more or less picked arbitrary names for them. File names may be just about anything but can not contain more than 55 characters. It is normally a good idea to make the name meaningful or somewhat descriptive of the file's contents. It is also wise to avoid certain characters in names which have special meaning elsewhere. Some examples of this are the '+' character, which is used to specify options, and the characters '*', '?', and '[' which will be described shortly. There are other special characters you will run into as time goes on, but for the most part, these are the ones to avoid. Some common sense can also be put to good use when assigning file names. For example, if you are writing a book, you may want to make each chapter a different file for easy reference. Naming the files "chapter1", "chapter2", "chapter3", and so forth, would keep them neatly organized. If you wanted to list all of the chapters you could type:

```
++ list chapter1 chapter2 chapter3 ....
```

but as you can see, if your book contained 25 chapters, this would be quite tedious. An easier way to accomplish this is by typing:

```
++ list chapter*
```

The '*' tells UniFLEX to look for anything in this part of the file name. Since the string "chapter" was specified, only file names which start with "chapter" will be looked at, but the names may end with anything. In this way, all of the file names we want will be provided to the list command for processing. It should be noted that once the qualifying file names are found, they are sorted alphabetically before being passed on to the command. In this case, "chapter1" will come before "chapter2" and so on, just as we want! This feature is not particular to the "list" command but is system wide. We could have just as easily typed:

```
++ page chapter*
```

to get a page formatted listing of the chapters. As another example:

```
++ list *
```

will display the contents of all files in your directory. Something just as easy, and very, very dangerous is:

```
++ kill *
```

which will delete all of your files! Do not experiment with this one, it really works! Another example of the name-matching feature is the following command:

```
++ list *file*
```

which will list all files which contain the string "file" as any part of the file name. The asterisk is very useful, and you should learn to make good use of it. There are some other name-matching characters available. One of these is the '?' which will match any single character. For example:

```
     ++ list test?
```

will list all files whose names start with the string "test" and have exactly one additional character in the name. The command:

```
     ++ list ?
```

will list all files whose names consist of a single character. There is one more name matching facility which allows you to specify a class of characters. For example, if you wanted to list chapters 3 through 7 and chapter 9 of your book you could type:

```
     ++ list chapter[345679]
```

The square brackets inform UniFLEX to match any of the characters between them. Another way to do the same thing is:

```
     ++ list chapter[3-79]
```

A range of consecutive letters or digits may be specified by separating them with a hyphen. This feature can be quite a time saver!

Now that you understand the basics of file names we will look at them a little closer. As mentioned earlier, all of your files are in a directory, which is nothing more than a collection of files. The system has many directories in it. Each user has a directory specifically assigned to him or her. This directory is called your home directory and usually has the same name as your login name. Yes, directories have names, just as do regular files. The fact that you have your own directory, different from another user´s, is why you can be assured that typing "list myfile" will list your file called "myfile" and not someone else´s. This implies that there can be more than one file with the same name in the system, as long as they reside in different directories. Whenever you create a file, it will be placed in your current directory, unless you instruct the system otherwise.

The directories of the system are arranged like a family tree. All directories have a "parent" directory with the exception of the "root" directory which forms the root or base of the tree. To get an idea of this structure, type the command "path". You should get something similar to:

```
     ++ path
     /usr/your-name
```

The path command prints the complete "path name" of your current directory. A path name, is the path you need to take through the tree to get to a directory or file. In this example, you must start at the "root" directory which is designated by the first ´/´ in the path name. Following is the name "usr" which is the name of a directory in the root directory. In this case, the root is the parent directory of the directory "usr". The next ´/´ character is simply a path name separator and is only a convention used by the system. Only a leading '/' represents the root. The next item in the path is "your-name" which should be the name of your directory (the same name as your login name). Its parent is "usr". To get a feel for the path name, type:

```
++ dir /usr/your-name
```

Of course you should substitute your actual user name for the string "your-name". This instructs the system to display the contents of the directory specified, in this case, your login directory. The results should be exactly the same as typing "dir" without a name since this command displays your current directory by default.

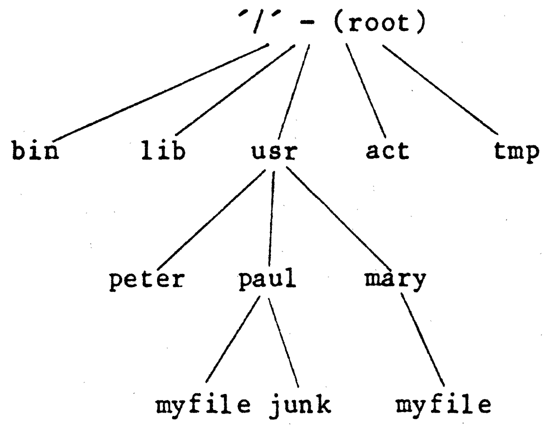
You can verify your path through the tree by first typing "dir /" to display the root directory. You will get back something like:

```
++ dir /
act          bin          dev          etc          lib
tmp          usr          usr2
```

The files you will see may vary slightly since they are system dependent. The key here is that you will find a file named "usr" which was the second element of your home directory's path name. Now enter the command "dir /usr". You should see a list (sometimes quite long) of all of the various user names on your system. Among the names you should find your own. This completes the path! As another example of path names, type:

```
++ list /usr/your-name/myfile
```

This should display the contents of your "myfile" (assuming it still exists). Since a picture is worth a thousand words, and this manual should be kept brief, here is a pictorial representation of the directory tree structure.



This is just a partial picture but you should be able to get an idea of how the different names are related. Notice that the file "myfile" in "paul" is completely different than the file "myfile" in "mary".

The two file names "." and ".." were mentioned previously. These are primarily for convenience. If you remember, the file "." represents the directory in which it is contained, and the file ".." represents its parent. If you type the commands "dir ." and "dir .." you will see how these work. Remember that the directory "usr" is the parent of your home directory.

Now that the directory structure is becoming clear, let´s look at some things we can do with it. If you wish to look at someone else´s directory, you simply need to specify the path name of the directory to "dir". As an example:

```
     ++ dir /usr/mary
```

will display the contents of mary´s directory. To copy a file from mary´s directory:

```
     ++ copy  /usr/mary/file  yourfile
```

The last two examples assume that you have permission to look at this directory and permission to read mary´s file. If you remember back to the long listing obtained from dir using the "+l" option, you saw the characters "rw-rw-" associated with each file. The full set of characters you will run into is

```
     rwxrwx
```

which represent the permissions associated with a file or directory. The first three characters are the permissions for the owner of the file and represent "read", "write", and "execute" permission respectively. The second set of three characters represent the permissions for all others. If you want to copy or list someone else´s file, you will have to have "read" permission in that file. If you want do a "dir" on someone else´s directory, you will need read permission for the directory. In this way, you may protect your files from others by setting the permissions as desired. The command to set or change permissions is "perms" and is covered separately in the manual <u>UniFLEX Utility Commands</u>.

As the number of files you collect grows, you will probably find your directory getting quite cluttered. UniFLEX offers the ability to create subdirectories in your home directory to help organize your files. For example, when writing your book, it might be desirable to create a directory which will contain nothing but your book. To do this, try:

```
     ++ crdir book
```

You will now have a subdirectory in your home directory named book. If you now execute "dir +l" you will see the name "book" listed. Notice that after the size number (the 1 right after the name) there is the character ´d´. The ´d´ informs you that "book" is not a regular file, but instead a directory. If you execute:

```
     ++ dir book
```

you will see that no files are displayed, showing that the directory is empty.

From your home directory, it would be necessary to type a path name to specify a file contained in your new subdirectory "book". For example:

```
     ++ list /usr/your-name/book/file-name
or
     ++ list book/file-name
```

Both of these will give the same results. The first example specifies the entire path name. Since the first character in the path name is ´/´, which represents the root, the system will start the file name search at the root, then go to "usr", then "your-name", followed by "book", and finally to the file. The second example will start in your current directory since the root directory was not specified. As you can see, this second form is much more convenient. It is also possible to change your current directory. If you are going to access a lot of files in "book", it would be even more efficient to change directories. For example:

```
     ++ chd /usr/your-name/book
or
     ++ chd book
```

will change your current directory to "book". Now if you type "dir" without specifying a directory name, you will be shown the contents of "book". You can also execute "dir .." which will display the contents of your home directory since it is the "parent" directory of "book". Feel free to create as many directories as needed to keep your files organized. This directory structure is one area which makes the system so efficient and easy to use.

You can also delete a directory if you decide it is no longer needed. You can change back to your home directory by typing

```
     ++ chd ..
or
     ++ chd /usr/your-name
```

If a directory is empty (no files contained in it except for the two special files "." and ".."), you may delete it with the "kill" command:

```
++ kill book +d
```

Notice that an option was specified ("+d") which tells "kill" it should delete directories. Without the "+d", the directory would not be deleted.

Now that you have a good understanding of files and directories, let´s look at some more advanced ways of executing commands. The program which is running and accepting your commands is called the shell program. It is the one which is printing the "++" on your terminal and issuing your entered commands to the system. The shell program will let you enter commands in a variety of ways. For example:

```
++ date; list myfile
```

will execute the date command, and when it is finished, execute the "list" command. The semicolon (´;´) tells the shell program that there is another command following. You may string as many commands together as you desire using the semicolon.

Another feature of the shell program is called I/O redirection. The term "I/O" refers to "input/output". As you noticed, most of the commands you have executed so far produce some sort of printing or output on your terminal. It is possible to redirect this output into a file. For example:

```
++ who >outfile
```

will not display anything on your terminal because the output has been redirected to the file named "outfile". The ´>´ is a special character to the shell program which causes it to take all standard output from the command it is going to execute and put it in the file specified. As another example:

```
++ list myfile testfile >junk
```

will put the contents of the files "myfile" and "testfile" one after the other in the file named "junk". This is a method of concatenating two or more files.

We can also group several commands together and put all of their output into one common file.

```
++ (date; who) >>status
```

The parentheses group the two commands together so that both of their outputs go into the file named "status". Without them, only the output from the command "who" would have been redirected into the file. Notice that in this example, the redirection symbol was ">>" instead of '>'. This mechanism provides essentially the same results with one exception. The '>' symbol will always create a new file when it operates, so if a file by the same name previously existed, its contents will be deleted before the command is run. The ">>" symbol will not delete the file if it exists, but will append the new output to the end of the file. This can be demonstrated by entering

```
++ (date; who) >>status
```

several times and then list the file "status" to see the results. Input to a command can also be redirected. As an example, the text editor may be used to make a variety of changes to a file. Let's suppose that the same set of changes had to be made to an entire set of files. It would be tedious to have to repeat the same operations for each file involved. Instead, we may create a file which contains all of the editor commands to be used, entered exactly as they would be entered to the editor. Let's call this file "script". We can now type the command:

```
++ edit  some-file  <script
```

The argument which starts with the "<" tells the shell program to send the contents of the file named "script" to the editor to be used as input, as opposed to using data entered from the keyboard. This demonstrates input redirection.

In the previous example, the files we may be editing with the script file may be quite large, and therefore, time consuming. Entering the command as shown will mean you will have to sit and wait for the command to finish. An alternative is to type the command as follows:

```
++ edit  some-file  <script &
```

Notice that the only difference is the ampersand character, '&', at the end of the line. The ampersand tells the shell program that it should not wait for the command to finish, but should immediately issue the prompt. This form of command execution is called background execution. When a command is run in the background, the shell program will report back with a line similar to:

```
** T2451 running.
```

The 'T' stands for "task" and the number following is the "task identifier" number, known as the task ID for short. When the command finally does finish, the system will report back with a similar line:

```
     ** T2451 - ended.
```

to signify that the task has completed. If you decide you do want to wait for the command after all, typing:

```
     ++ wait
```

will cause the system to act exactly as if the ampersand were not specified initially. In other words, the shell program will not accept any more commands until your background task has completed. It is possible to execute more than one background task at a time. If you get several going, the "jobs" command will give you a list of your current background tasks, their task IDs, and the date and time they were started. If you need to stop a task running in the background, you will quickly find out that the control C described earlier does not work. The "int" command is used for this purpose. It can be used to interrupt or terminate a background task. As an example:

```
     ++ int 2451
```

will interrupt the background task number 2451.

Another very powerful feature of the shell program is pipes. Pipes are used to connect small programs which are usually "filter" type programs. A filter is a program which is provided some form of input, alters the data in some way or another, and then outputs this changed data. The "page" command is an example of a filter. As an example:

```
     ++ page file1 file2 file3
```

will list each file starting at the top of a new page. Suppose you want the page formatting that the "page" command performs, but you want the files all run together, as if they were one file. One way to accomplish this is as follows:

```
     ++ list file1 file2 file3 >temp
     ++ page <temp
     ++ kill temp
```

This sequence will concatenate the three files (using "list") and put the output into a file called "temp". The next line will run the "page" command, which will get its input from the file "temp" (input redirection). Finally, since we don´t really need the "temp" file, it is deleted using "kill". This is fine, but there is a much easier way to accomplish this:

```
     ++ list file1 file2 file3 | page
```

The vertical bar character (the '^' character may also be used) sets up a pipe and tells the system to take the output from the "list" command and use it as input to the "page" command. The page command will then output the information directly to the terminal. The fact that the data are routed through several commands is where the name pipe comes from. Another example of a pipe is

```
     ++ dir +l | page
```

which will give you a page-formatted directory listing. Remember that the "+l" specifies the long listing. Pipes may of course be run in the background.

```
     ++ command1 | command2 &
```

Again the ampersand character is used, as described previously.

Another feature of the shell program is the "startup" service. You may find yourself repeating the same sequence of commands every time you log in to the system. As an example, you may run the "who" command to see who else is on the system, followed by the "dir" command to check the contents of your directory. This can be done automatically by creating a file named ".startup" in your home directory. This file should contain the list of commands you want executed. For example:

```
     ++ edit .startup
     New file being created
         1.00=who; dir
         2.00=#s
     ++
```

You now have a file which lists the two commands you run each time you login. The next time you login, the system will read your ".startup" file and execute the commands you have listed automatically!

The shell program can execute other files containing lists of commands as well. Create a file in the following manner:

```
     ++ edit dw
     New file being created
         1.00=date
         2.00=who
         3.00=#s
     ++
```

To execute this file, simply type:

```
     ++ shell <dw
```

which runs the shell program (after all, it is just another program) but gets its input from the file named "dw" rather than the terminal. Both of the commands, "date" and "who" will be executed, just as if you had typed them in. We can make this even more convenient by telling the system that the file "dw" is to be made executable. This can be done by:

```
     ++ perms u+x dw
```

which sets the "execute" permission for the user (owner) of the file "dw". Now all you need to do is enter:

```
     ++ dw
```

and the date and "who" information will be displayed.

Quite a bit has been presented in this introduction. Hopefully you have tried most of the examples. If everything is not clear, don´t worry. Reread the parts you had trouble with, and experiment. The best way to learn what the system can and cannot do is by experimenting. In a short time you will probably be calling yourself an expert!