```
=============================================================
|                                                           |
|        #         # # #        # # # #    #         #      |
|      #   #       #     #      #            #     #        |
|    #       #     #      #     #             #   #         |
|   #         #    #       #    #              # #          |
|   #         #    #     #      #              # #          |
|   # # # # # #    # # #        # # # #        # #          |
|   #         #    #            #            #     #        |
|   #         #    #            # # # #    #         #      |
|                                                           |
|                 VERSION 1.0   MARCH 1980                  |
=============================================================
                 Copyright P.J.R. Boyle 1980.
```

PREFACE

Apex is a powerful and general operating system. The ways in which it can be used are almost infinite. You will need some patience at first since Apex is not a simple application program which can lead you by the hand. In some configurations Apex will be used with a high level language such as XPLO, which will hide the messy details of Apex for you. However the basic Apex package is intended for assembly language programmers and that is the focus of this document.

Any documentation of such a system must ne cessarily be rather terse and self referent. This manual supplies the essential information you will need to use Apex. A lot of this information is not critical when Apex is used in the simplest way but is essential for more complex operations. Thus this manual will need several readings. First to get the overview, then to get the basic information you need to begin to use Apex and, later, to extract the detail you need for some specific task.

Eventually you will need to read and understand the listings provided with this manual in order to use the full potential of Apex. Note that these listings are supplied for pedagogic purposes only. The specific code you recieve with Apex may be a later revision or may be for a different system configuration. The principles will remain however.

CONTENTS

INTRODUCTION

Welcome to Apex. You now have at your disposal an entirely new organization of your Apple II which will make many new things possible. Apex will make your Apple perform in ways which you have previously only found in much larger machines. Best of all, Apex will do this without tying up large chunks of memory, without limiting the things you can do with your Apple, and without a continual machine-time overhead.

Apex is a programmer´s operating system. It is designed to provide a powerful program development environment while retaining a general purpose overall structure that will be compatible with most specific application tasks.

Apex has a multi-level structure. Some of Apex is always in memory while other parts of it will share memory with your program in the same way some large machines implement "virtual" memory; that is by swapping portions of memory to and from the system disk.

Apex has been designed for simple program interface. In principle no program need know anything about Apex unless it wants to use Apex facilities. In this way, almost any program may be run under Apex. The exception, of course, is a program that was written to use the facilities of some other operating system, such as Apple DOS.

The most fundamental service Apex provides is the organization and maintenance of your disk space. It does this by partitioning the disk space into "units" and "files".

Units are large, fixed areas of disk storage space which are allocated in part in correspondence with external constraints, and in part in accordance with convenience factors. On small systems, a unit will typically correspond to a physical floppy disk.

Files, on the other hand, are sections of a unit which are variable in size and have names. Apex provides you and your programs with access to files by name and to units by their number.

Another major service Apex provides is a modular and global mechanism for accessing your peripheral devices, such as terminals, printers etc. Apex contains a set of modular programs called device "handlers". Each device has a handler, which is the only piece of code which needs to know about the specific details of the device. This means that programs under Apex need not know about the specific details of the printer you have etc. Normally your standard version of Apex will contain handlers for your peripheral devices as well as for the Apple keyboard and screen (together known as the "console" device), and for the disks you have.

A word about languages is in order here. Unlike Apple DOS or Apple Pascal, Apex is not imbedded into any one langauge. Instead, it provides a general overall structure into which any language can be placed. This has the important advantage that you don´t have to re-learn your machine every time you change languages. The root language of Apex is, of course, Assembly language. Other languages have been implemented under Apex, notably XPLO and FOCAL. They are probably available from the same source you obtained Apex. Apple´s BASICs can be run under Apex if you insist, but of course those functions which are DOS related will not work. UCSD Pascal is its own operating system and will not run under anything.

In every software project there are certain design decisions which must be made. In Apex some of these decisions have been made differently from most other operating systems. Loosely speaking, the reason for this is that Apex has a different focus, and so the priorities come out different.

For one thing Apex uses contiguous files. That is, files on an Apex disk reside in sequential block numbers. This makes multi-block disk transfers much faster. Apex can load a program into memory, or a source file into an editor, at speeds that are impossible with the linked block structure used in Apple DOS. The cost is that in Apex files must be competely re-written when they are modified. Copying a file when you modify it is a good idea anyway, since this makes the process non-fatal in case of disk errors.

For another thing, Apex has a very small resident portion. The fundamental advantage of this is obvious: You have more user memory. An un-obvious advantage of this arrangement is that, since the basic structure of Apex does not define the run-time environment, Apex can support almost any type of task. It is possible to build a foreground-background, real-time or data-base management environment for your programs within the same basic structure. The penalty is that the run-time code your programs need does not come as a basic part of Apex. It must be loaded with each program, either as a separate system module, or as a part of the program itself.

Another tradeoff exists in the area of ease of learning versus the amount of typing you have to do. Apex is intentionally cryptic on input and verbose on output. This can a little confusing at first, but soon you will become very familiar with it, and the cryptic input will greatly speed your work.

A related matter is the issue of protections. In Apex many protections are built in, such as backup files, backup directories, read-after-write options, and volume number checks. However you are the boss, and its your system, so Apex will allow you to do anything if you insist, no matter how fatal the operation may appear to be.

BOOTING APEX

If you have a standard Apple II or Apple II+ with a standard Apple II mini disk drive, the Apex system disk you received with this package should boot, in the same way DOS boots, on the disk you have connected as drive 1 on slot 6.

The first things you will want to do upon booting Apex for the first time are:

1) Study this manual until you have a good idea of what Apex is and how it works. Otherwise there is no chance that you will succeed in using it.
2) "Permit" your disk drives - see the section on unit allocations, and the BOOTER utility.
3) Make some working disks from the master disk. You will use the MAKER and DUPDSK, EXCH or COPY utilities.

BOOTING ON NON-STANDARD SYSTEMS

The bootstrap on the standard master system disk assumes that the standard Apple bootstrap procedure will work on your machine. It may not if you have a different bootstrap ROM on your disk controller (such as if you have the language card) or if you have a different type of disk drive. In this case you need a special bootstrap. If one is not available you must boot the other disk you received, which is an almost standard Apple DOS disk. On this disk you will find a "binary" file called APEXBOOT. You must contrive a method to BRUN this file. It loads into memory at address $3000 and is $800 bytes long.

When APEXBOOT is started it prompts you for an Apex disk and loads Apex from that disk. APEXBOOT does not itself contain any of the code for Apex. It reads this information from the Apex system disk in the same way that the normal bootstrap process would have read it. In this way the Apex bootstrap maintenance programs still apply.

DISK ORGANIZATION

The track, sector, slot, and drive organization of normal Apple disks is not used in Apex because it is not general enough.

Instead, disk storage is organized into blocks, files and units. These are translated into the appropriate physical entities, for the specific disk drive in question, by the disk "driver" within the Apex resident code. The standard Apex system has a driver which supports the standard Apple 5" disks as well as 8" single density disks using a controller made by Sorrento Valley Associates. Drivers for other disks can replace the standard driver.

All of your disk space is ultimately organized into blocks of 256 bytes each. A unit is manipulated as a fixed sequence of applicable block numbers. Block numbers begin at zero and go up in sequence to the size of the unit minus one.

WRITE LOCKING DISKS

Apex is continually accessing the disks. Thus, write-locking disks will stop you from doing almost anything. However, it is possible to run, read, or copy files from a write-locked disk.

UNIT ALLOCATIONS

In the standard Apex configuration, there are eight possible units, numbered 0 through 7. These units are assigned to physical disks through tables in the disk driver code. There is a default setting of these tables that exists in the driver as distributed. The resulting assignments are listed in the table below. In addition to the disk driver tables, there is a "permit" byte in the system page which tells Apex which units are extant on the current system. If the standard setting of the tables includes the drives you have, then all that is neccessary in order to enable or disable your drives, is to change the permit byte at $BF51. When you change this byte, or the driver tables, you can make the change permanent with the program BOOTER. In the permit byte, each bit corresponds to a possible unit. Bit 0, the least significant bit, enables unit 0, bit 1 enables unit 1 etc.

Note that unit 0 has a special significance. It is the unit on which the distributed Apex disk will boot. It is also the unit to which Apex will return if an error should occur while accessing the current system unit.

UNIT #	SLOT#	DRIVE#	PERMIT BIT#	TYPE
0	6	1	0	Apple disk II
1	6	2	1	Apple disk II
2	5	1	2	Apple disk II
3	5	2	3	Apple disk II
4	7	1	4	SVA 8" disk.
5	7	2	5	SVA 8" disk.
6	7	3	6	SVA 8" disk.
7	7	4	7	SVA 8" disk.

STANDARD UNIT ASSIGNMENT TABLE

UNIT ORGANIZATION

A unit has some blocks (0-8) reserved for bootup code etc. Following these blocks is a unit directory (9-12) which describes the files on the unit. On most units, a backup directory follows the main directory (13-16). The rest of the unit (17-) is available for file storage. A bootable disk requires that the first few blocks of file space (17-25) contain resident code in a special system file.

A standard Apple mini-disk has 455 blocks on it of which 438 are available for files.

SYSTEM UNITS

In Apex, a unit may be either a system unit or not. At all times there should be at least one valid system unit on line, so, if you have only one disk drive, most of your disks will be system units. If you have more than one disk drive, you will probably have only one system unit on line at a time. A system unit is distinguished from a non-system unit by the existence of the files with the SYS extension. These files can be deleted from a system unit to make the unit into a non-system unit.

FORMATTING DISKS

Apex disks must, in general, be formatted before they can be used. This process is a property of the individual disk and drive you have, so Apex leaves this process as an external function to be completed by the process described by the disk drive supplier.

The DOS disk you received with Apex has one important detail which is different from most DOS disks. The DOS "INIT" process, which formats an Apple II 5" disk, when performed from this disk, will format a disk with a special sectoring which allows Apex to run at the maximum speed.

Note that the Apple DOS "INIT" process is something unique to the Apple mini-floppies. It has nothing to do with other disks, the Apex INIT command, or to the Apex MAKER process which must be performed separately.

FILES IN APEX

TYPES OF FILES

At this time, there are three main types of files handled by APEX. They are system files, save files, and text files.

System files contain the Apex system itself and have a unique format determined by the operating needs of Apex.

A save file is a memory image of an executable program. The program is saved on a unit along with run-time information, such

as starting address, exit address, error address, and default information.

A text file is a serial sequence of ASCII characters. The file can be composed of any ASCII text characters except Control-Z ($1A). Control-Z is used as an end-of-file mark.

FILE SPECIFICATIONS

When written out in full, an Apex file specification consists of a unit number, a file name, and a file extension. For example:

```
        2:NICENAME.TYP
      .     .        .
    .       .          .
unit    file name    extension
```

The legal partial file specifications can be determined from the syntax diagrams. In many cases a file specification can be abbreviated to only one of the three parts, or even to just the colon character. A number alone, with no colon, will be taken as a unit specification, not a file specification, whenever both options are possibilities.

The unit number determines the unit which has the file, the name is an arbitrary name you assign (except for SYS files), and the extension is a file type descriptor.

In cases where a file specification is intended to specify more than one file, such as in copy or delete operations, a "wild" or "fuzzy" file specification can be used. Any character in the name or extension part can be replaced by a question mark character which will match any character in that position. The entire name or extension part can be made fuzzy by substituting an asterisk for that part of the specification. Thus "*.*" means all files, while "*.X??" means all files whose extension begins with the letter X.

ILLEGAL FILE NAMES

Apex file names can be up to eight characters long. The first character may not be a digit.

You are prevented from creating files whose names contain the slash, asterisk or question-mark characters. Also you may not create files with the extension BAK.

STANDARD EXTENSIONS

File name extensions are used to distinguish the different forms of the same information. For example the same program can exist as a source file (P65), a backup source file (BAK), an assembled binary file (BIN), and an executable image (SAV).

The assignment of file extensions is up to you. You may freely invent and use extensions at your whim. However, there are certain special extensions and certain conventional extensions which will occur in the defaulting process. The special extensions are SYS and BAK. The basic conventional extensions are SAV, P65, and BIN. Depending upon the software you are running under Apex you may come across other extensions like XPL, I2L, FCL, LST, BAS, TMP, or DAT.

The special extensions are:

SYS	These are special files which exist on a system unit and have special properties. See later (pg. 12).
BAK	These are files which contain a previous revision of a text or data file.

Some conventional extensions are:

SAV	These are files which contain a special form of memory image which is directly executable by a run

command.

P65 These are Assembly language source files.

BIN These are the assembler output and loader input files.

USING FILES

ASCII files generally contain information that is to be processed in one way or another. The information is usually processed by a program which exists in a SAV File.

```
=========       ==========      =========
! output !<<<<<<!  pro-   !<<<<<<! input  !
!  file  !      !  gram   !      !  file  !
=========       ==========      =========
```

As the information is processed, it passes from the input file through the program to the output file.

The input file can be any file that exists on a unit. Since the information is read from the input file, it must already exist on some unit. The program can be any program that processes information and has been saved as a SAV file. The output file is a vacant area on the designated output unit that is set aside by the operating system. The vacant space is filled by the freshly processed information. It is not neccessary for the program to have both an input and an output file. Depending upon the nature of the information processing, the program may require only an input file, only an output file, or many files.

OPENING AND CLOSING FILES

The process of setting up input and output files is called "OPENING" the files. The Apex run and OPEN commands provide a simple means for opening the basic input and output file required by many programs. As in:

```
        ASM FROG
or
        OPEN PIG<DOG
```

The system resident function KSCAN can be used to find other files on a unit (pg. 37).

These files are "setup" by Apex and the information about them is placed in the input and output file descriptors in the system page. Programs and handlers that need to use the files can retrieve this information and use it to control their access to the disks.

A file setup by Apex will have a status byte equal to one. If no file was setup by Apex the associated file descriptor block will have a status of zero (pg.34).

Initially Apex will assign an output file to the largest remaining empty space on the designated unit. The program which writes information into the file must reset the ending block number to the actual last block used, and then mark the file closed, by setting its status byte to $FF. When the Apex Exec regains control, it will correct the directory of the unit to reflect the actual size of the file.

If the files are to be accessed in a byte serial fashion, as is the case with many programs, assemblers for example, then the files can be treated exactly like any other byte I/O device and accessed through the byte I/O device handler (device 3). In this case, the program need not do anything special at all.

As with any device, the program must open the device for input and/or output, read or write bytes to the device and close the device before it runs to termination. These operations are all standard functions which every device handler must be able to perform.

If a file is written, but not closed, it will be discarded unless the Apex command CLOSE is used before any new files are opened.

BACKUP FILES

Whenever a file is modified, a new copy is made, and we have a choice. We can either delete the old version, or we can rename it so that the same name does not occur twice in the directory. In Apex, the choice is yours.

In some instances, the Apex will simply delete the original file, leaving only one file with the name. In some instances, it will change the extension of the original file to "BAK". In this case, you always have at least one old version of the file in case of disaster. The system wide switch BACKUP turns the option on or off. It is changed by the DO and NO commands to Apex.

Even when we have the backup switch on, we do not want every file name duplication to produce backups. Generally we only want to backup source files. To control this, every program which produces an output file has its own local backup switch. This switch is changed by the SET utility. In order for backups to be produced, both the local and the system wide backup switches must be on.

Apex protects BAK files by preventing you from changing them. In order to work with them as normal files, you must rename them.

SYSTEM FILES

In Apex there are special system files with the SYS extension. They are:

RESCOD.SYS
: This file contains the resident portion of Apex. It is only required on bootable system units. It is 9 blocks long and must begin in block 17.

SYSTEM.SYS
: This file contains the command executive. It is required on all system units. It is 65 blocks long and can reside anywhere on a unit.

SCRATCH.SYS
: This file is the space Apex will use to save the part of memory that the command executive uses. It is required by several system functions and will normally exist on every system unit. It is 65 blocks long and can reside anywhere on a unit.

DEFAULTS

Apex has a powerful default mechanism which will allow commands to be extremely abbreviated. These defaults may take a little getting used to at first, but in time you will develop a feel for just how much you really need to type to get Apex to do what you want. The syntax diagrams in this manual will help you determine the range of possibilities.

Defaults in Apex are based upon a few basic concepts which are described here.

Apex has a "system" unit and a "task" unit. The system unit is initially the unit you booted from, but can be changed by the SYSTEM command. The task unit is the unit number attached to the default file name, which can be changed with the DFILE command.

In operation, Apex will use the system unit for storing such things as default information, the date etc. It will also, unless told otherwise, look on the system unit for programs you request in a run command. Hence, the system unit is where you normally keep your commonly used programs such as editors and assemblers. The task unit is the default for all other operations. For example, input and output files will default to the task unit. Hence the task unit is where you keep the program you are working on.

Apex has a system wide default file specification which will be used whenever no other file name is specified. The default file has a unit and extension attached to it. The unit will be the task unit and the extension will be used when all other sources of a reasonable extension have no suggestions to make.

In general, the context of a command will suggest an extension which has a higher priority than the default extension, but a lower priority than an explicily given extension. For example, the GET command will suggest an extension of SAV for its file argument.

However, extensions are for your convenience, not Apex´s, so you may invent and use any extensions you please. This leads to a possible danger. If you spell it out, Apex will happily try to run a text file or edit a runable program image.

GENERAL DEFAULT STRUCTURE

Basically, the default structure automatically provides file names whenever the user omits them. The system default file is used to control the current working file. If an OPEN or run command is entered without an argument and followed by a space, the system default file name will be taken as the input and/or output file for that command.

Each saved program can automatically control its input and output extensions. This enables programs like the assembler to specify BIN for the binary output file, or the XPLO loader to specify I2L as its input file extension. These suggested extensions are kept in the program specific part of the system page and can be set up using the SET utility.

When these two systems of defaults are used together, they eliminate most of the typing during program development.

For example, let´s say that you are writing a machine language program to catalog your stamp collection. You have decided to call the program "STAMP". Since this file will be the active working file, you will set the system default file name to this file with the DFILE command. Assuming you want unit 1 to be the task unit, you could type:

```
        DFILE 1:STAMP.P65
```

Generally, you will be editing, then assembling, then loading and finally saving the file. Initially you must make the file for the first time. This can be done in several ways, but simply typing MAKE followed by a space and a return will do the job.

If you then type the name of the editor (probably EDIT) followed by a space, the editor will take STAMP.P65 as it´s input and output file.

When you are finished editing, you are ready to assemble. Running the assembler will read STAMP.P65 as it´s input file. But, because the assembler has a special extension for it´s output file, it will create STAMP.BIN as the output file.

As soon as the file is assembled, you will probably want to load. Typing LOAD followed by a space will set STAMP.BIN as the load file for the loader. No output file will be set, because the loader default indicates that no output file is needed.

The loader will return to the Apex Exec. At this point, you can use the SWAP or START commands to test the program or you can save the program with the SAVE command. Note that, unless the system page properties of the program were set up as a part of the loading process, the START command cannot be used yet, and the SAVE command will need to be told what areas of memory to save.

After the save, SET (space)(return) can be used to setup the program´s run parameters.

The default structure is set up so that it can be overridden in several different ways. Generally, the defaults can be overridden by explicitly spelling out part of the input or output file specification. Here are some examples:

EDIT A.FIL<A.FIL	Opens "A" for input and "A" for output.
EDIT B.FIL<A.FIL	Opens "A" for input and "B" for output.
EDIT B.FIL	Opens "B" for both input and output.
EDIT B.FIL<	Opens "B" for output only.
EDIT B	Opens "B" with the default extension.
EDIT <B.FIL	Opens "B" for input only.
EDIT(SPACE)	Opens input and output default files.
EDIT .TMP<.FIL	Opens the file with other extensions.
EDIT 2:TEMP<:	Opens a new file for output, same input.

Refer to the description of the OPEN and run commands for further discussion (pgs 38,42).

SWAPPING

An Apple II has a relatively small memory space. Even a fully expanded system is only 48K. A typical operating system runs in 10-20K, so that even with fully expanded memory, the operating system uses up one third of the memory.

Apex deals with this problem by sharing the memory between the user program and the operating system. If the program is smaller than about 20K, the operating system and the user program can fit in main memory without any conflict. If the program grows larger than 20K, it is allowed to over-write part of the Apex Exec. Whenever it is needed again, the Exec is simply reloaded from disk.

In the situation where a large program and the operating system must co-operate, the Apex Exec is swapped in and out of memory as needed. The swapping operation involves saving the resident program in a scratch area on the disk and then reloading the Exec. Thus, an exact copy of the current state of the program can be preserved while the Exec is in memory. Since Apex is fast, the entire swapping operation takes only a few seconds. The swapped out program can be restarted or can be manipulated by the operating system. Using this technique, every program can use nearly all of the computer´s memory.

WHEN YOU SWAP AND WHEN YOU DON´T

Even the few seconds of swap time is unnec essary most of the time, since you are typically re-entering Apex without caring about the previous memory content. So the "normal" entry to Apex is the REENTER point, which does not save the current memory content. However, if you want to preserve the current memory content, for later use, or because you want to make a SAV file out of it, you must swap. Also, error conditions should swap whenever possible so that the debug process is simpler. Thus, there is a SAVER entry point to Apex which is taken whenever a program is aborted with the Control-P key or when you use Control-Y to re-enter Apex from the Apple ROM monitor. It is also taken as the standard exit of loaders which setup a memory content for future use. Apex entry points are discussed further under that topic (pg 36).

You can return to a swapped out memory content with the SWAP and START commands to Apex. You can save a swapped out memory content in a file with the SAVE command. You can restore a SAVEd memory content with the run and GET commands.

RUNNING WITHOUT SWAPPING

The Apex system files use a significant amount of space on an Apple mini-floppy. If you have only one drive you may encounter a program which simply cannot be worked with on a single system unit.

It is possible to go through a sequence of running programs which do not use the command excutive memory space without a valid system unit on line.

So, in extreme cases, with care, you can use a non-system disk to edit and assemble large programs. The loading process will require a system unit however.

The distributed assembler and editor are setup to not overlay the Apex Exec. This limits their capability somewhat. You can set them up to use more memory, by changing the appropriate system page parameters (pg. 33). However then they will have to be be run from a valid system unit.

DATES

One of the most reasonable ways to keep track of your work, particularly on floppy disks, where numerous revisions of a program tend to get scattered over many disks, is the use of dates on files. Apex maintains a system date which it saves on the system unit and which it will attach to every file as it is created or when it is modified. It is a very good idea to get used to maintaining the system date right from the start. The DATE command is used to change the system date, and the /L switch on the DIRECTORY command is used to see the dates attached to files.

ERRORS

Apex is verbose and quite clear in most situations. Its messages and this manual should be sufficient to follow its normal operation.

However, in the case where an error occurs in the runtime system, the error message is cryptic. You get a question-mark and a number alone on a line. Normally the number is 3, which indicates an I/O error. The most common I/O error is an attempt to access a disabled disk unit, such as a drive with the door open, or with no disk in it. Any other number indicates an unreasonable condition in the runtime system and probably means that that area of memory has been changed and you must reboot.

APEX SYNTAX

This manual will not discuss the Apex syntax in any detail. The examples should be all you need. However a few points must be made here.

Apex commands are usually one line long. The RETURN key ends the command. Until that time the line is being processed by the console handler and so the command editing features of Apex depend upon the handler you have. See that section for details (pg. 29).

If a command line is given to Apex, and is incomplete in a determinable way, Apex will simply request more input, which results in a new flashing cursor on the next line.

Apex regards the slash character as the command line switch prefix. Hence, any slash character, and the character which follows it, are not a normal part of the command. There are also other special characters used to terminate programs etc. These are properties of the console handler and may vary. However there are fairly strong conventions for these, which this manual will assume at times.

Numbers given to Apex are taken as sixteen bit unsigned integers. The minus and period characters are not parts of numbers and will usually be taken as delimiters.

Numbers in Apex can be in decimal or in hexadecimal. If the number is to be taken as hexadecimal it must be prefixed by a dollar-sign.

SWITCHES IN APEX

COMMAND LINE SWITCHES

Some commands use a special feature called "command switches". Switches are special character included with a command that modifies the action of the command. Command line switches are preceeded by a FORWARD SLASH character (/). Any character following a by a forward slash will be taken as a switch character. Switches can be placed anywhere within the command line. For example:

```
DIRECTORY/L
```

SYSTEM WIDE SWITCHES

In Apex there are three system-wide switches which are changed by the NO and DO commands. These are the BACKUP, CHECK and PACK switches.

The BACKUP switch will allow or prevent the backup file feature. With BACKUP on the source files you are working with will not be erased until they are two revisions old. See the section on Apex files (pg. 8).

If the CHECK switch is on, Apex will check every output file to make sure it is readable before deleting or renaming any old version of the file that may exist.

If the PACK switch is on, Apex will take time out every now and then to move files on the disk to keep the empty space on the unit more optimal. Packing will only move the most recently closed output file so fragmentation of the empty space can still occur in time, but it takes much longer with the pack switch on than with it off.

SAVE FILE SWITCHES

Each SAV file which expects to produce an output file has three switches to control this file. They are the BACKUP, SIZE and KEEP DATE switches. These switches are maintained in the program specific portion of the system page and may be setup manually or with the Apex SET utility.

The SAV file BACKUP switch tells Apex whether this program produces an output file which is a revision of its input file. Progams which merely revise files can reasonably backup the old version of the file. An editor is an example of such a program. Programs, such as assemblers, which produce completely different output files simply leave the input file alone and so have their local backup switch off.

If the SIZE switch is on, Apex will require that the available space for the output file be at least as large as the input file. This is a useful protection for editors and other programs which tend to make files larger.

The KEEP DATE switch controls the date assigned to a file. If it is on, then the date on the output file will be obtained from the date that the input file had; otherwise, the file will be dated with the system date.

MEMORY USE

There is a diagram of apex memory use in this manual. It should be referred to in conjunction with this discussion.

At the most central level, there is a section of Apex which must be resident at all times. This section lives in the highest 4k of your machine and contains the basic system functions, the disk drivers, and the code to handle the keyboard and screen.

The highest page of your memory, the section from $BF00 thru $BFFF, is the central communications area for Apex. It is through this page that all programs running under Apex will communicate with Apex itself. This page is discussed in detail elsewhere (pg.31).

At the next level we have another 4K section of memory, called the system residual area, which normally contains Apex device handlers and the code for certain run-time system functions. When your programs are loaded, this section of code remains in memory and so its funtions are available to your programs. However, your program is free to overwrite this area with other information if required. This area of memory is also used as the place where other run-time utility packages for specific applications are placed. The normal contents of this area will be restored when your program runs to completion.

The next level, the Apex Exec, is overlaid onto a 12k section of memory below the system residual area. Programs which do not use this area of memory can leave the Apex Exec resident while they run by setting the SYBOMB flag in the system page to FALSE=0. If your program does use this area of memory, the SYBOMB flag should be set to TRUE=$FF, which will cause Apex to reload the Exec and the residual area when your program runs to termination.

Apex allocates page zero (hex addresses $0 through $FF) as follows. The first 32 bytes are considered very temporary and are never saved. Apex will frequently use the first six of these. The next 48 bytes are reserved for use by the ROM in your Apple and should not be used by normal programs. The remainder, from $50 through $FF is for use by your programs and will be saved with them in SAV files.

The first 80 bytes of the system page at $BF00 are a part of a user program. They are used to describe various properties of the program which Apex needs to know about. Therefore these bytes are also saved with a program. The SET utility can be used to modify these locations in a SAV file.

Every program which uses Apex files must pay some attention to the location of its input and output buffers which are defined in the system page. This is discussed in more detail elsewhere (pg.32).

Beyond these memory allocations Apex leaves the rest of your Apple alone. Thus the maximum possible flexibility is maintained.

OTHER MEMORY USE

The standard bootstrap process will use pages 3, 8 and 9 as well as the memory from $6000 upward.

The SAVER entry process will use memory from $A000 upward, saving $6000-$9FFF in the scratch area of the system unit.

The console handler will use page 2 as the input line buffer. Also, by using ROM routines, it will affect the areas of page zero between $20 and $4F.

The bootstrap process will setup the auto-start and Control-Y vectors in page 3.

Be warned that the Apple BASICs will use page zero locations other than those set aside for ROM use.

DEVICE HANDLERS

Apex has a device independant mechanism for accessing peripheral devices which can be made to communicate in a serial byte stream. Each such device has a handler. Each handler has a set of five standard functions which it must be able to perform. In addition, a device may have other functions which it can perform as well, but these will be different from one device to another.

Handlers in Apex are "plugable". That is, you can replace any handler in Apex by another piece of code without affecting anything except the funtioning of that device. This is how you tailor Apex to your specific printer for example.

Apex handlers are not sacred. You are free to change them at your whim. For example, if you are used to some line delete character other than Control-X then you are free to change the console handler accordingly.

Of course, handlers are a convenience, not a rigid structure. Your programs need not use the handlers; after all, this is your machine! Certain programs, in fact, will find the formalized structure of handlers a problem, and so will contain their own code to drive the device in question. The classic case is editors which interface with the the most complex peripheral of all - you. But note that if a program does use the handlers then it will be "device independant" and will be able to use any device that you can create a handler for. Since no system stays the same for very long, this is a major advantage.

On the Apple II, most peripheral devices are supplied with an on board ROM which is supposed to contain the code necessary for driving the device. If this mechanism is satisfactory for a particular device, then the Apex handler becomes simply a series of jumps to the appropriate slot address, $CX00. However, it is common for the ROM code to be oriented to some special purpose, other than the one you have in mind, and so the handler becomes a little more complex. Apex gives you this freedom.

HANDLER ENTRY POINTS

Each handler must begin with five jump vectors which occur in a defined sequence and perform specified functions. These are listed below. Every handler function must return with the carry flag reflecting the success or failure of the requested function. A set carry flag indicates failure. Data is passed to and from handlers in the accumulator and the Y register as neccessary for the specific function. A handler need not preserve any registers and should never be assumed to do so. Handlers may not change any system page information, and may not use any locations in page zero except the first six.

1. OPEN FOR INPUT ENTRY=0

This entry point must initialize the input side of the device. This entry point must always have been called before the INPUT entry is ever called. This entry should also check for the existence and readiness of the device to perform byte input. It should flush the input buffer if any.

2. OPEN FOR OUTPUT ENTRY=3

This entry point must initialize the output side of the device. This entry point must always have been called before the OUTPUT entry is ever called. This entry should also check for the existence and readiness of the device to perform byte output. It should reset the output buffer if any.

3. INPUT A BYTE ENTRY=6

This entry point will fetch a byte from the device and return it in the accumulator.

4. OUTPUT A BYTE ENTRY=9

This entry point will output the byte in the accumulator to the device.

5. CLOSE DEVICE ENTRY=12

This entry point will terminate access to the device. Output buffers, if any, are flushed. If the device can be powered up and down under program control, this entry should turn it off.

ACCESSING DEVICE HANDLERS

The standard method of calling a device handler to perform a function is as follows: The number of the device to be accessed is placed in the system page location NOWDEV at $BF5C. The X register is set to the number of the function to be performed. The Accumulator and Y registers are setup, if required, and a subroutine call (JSR) is made to the system entry point KHAND at $BFD9. KHAND will use the device handler table to dispatch to the correct entry point.

INSERTING DEVICE HANDLERS.

In order for KHAND to be able to find the device in question the base address of its handler must be placed in the device driver table in the system page. Device drivers can be anywhere in memory, but, in order to have them automatically loaded, they should be placed in the system resident or system residual area. The system residual area is saved in the system file SYSTEM.SYS when the INIT command to Apex is performed. The system resident area and the system page area containing the device driver tables is saved in the file RESCOD.SYS when the system utility BOOTER is used to rewrite the bootstrap.

STANDARD DEVICE HANDLERS

Every implementation of Apex should have the following device handlers:

DEVICE 0: The console, as a line by line device. This handler should provide a means for the user to correct the console input line before making the bytes available to the calling program. The open input entry should clear the input line. This handler should deal with the echoing of the input back to the output.

DEVICE 1: The console, as an unbuffered, non-echoed, character by character device. It should never give errors. This handler should deal with the trap characters (normally Control-C and Control-P) which can exit or abort the currently running program.

DEVICE 3: This handler provides access to the open Apex files as byte by byte streams. The open for input entry should reset the input file to its beginning. The close entry should write an end-of-file mark to the output file and close it. A file which is written but never closed should be discarded.

DEVICE 7: A null device. It provides endless end-of-file marks on input and swallows endless amounts of output. It never gives errors. Typically this device is used as a sink for unwanted output, such as an assembly listing which is not needed.

Most implementations of Apex will also have a printer handler as device 2 and many will have a serial RS232 port as device 4. Devices 5 and 6 are available for other peripherals.

THE CONSOLE DEVICE

This section gives some specifics on the standard console handler supplied with Apex. Listings are included, both to serve as an example handler, and so that the exact functioning can be determined.

Devices 0 and 1 in Apex are the standard "console" which consists of the Apple keyboard and TV screen. In some configurations, the console device will be a terminal accessed through a serial interface or an alternative TV display card. In these cases the console device handler may be different from the handler described here.

The difference between device 0 and device 1 is that device 0 is line buffered and automatically echoed, while device 1 is not.

The basic console handler, device 1, deals with keyboard input as follows: Most keys produce the correct 7 bit ASCII code one would expect. Note that, unlike the Apple ROM routines, the high bit of the character is not set. Certain keys have special functions.

The trap keys are as follows: Control-C will exit immediately through the current program's normal exit vector. Control-P will exit through the current program's abort exit vector.

Some keys are translated to other characters: Control-O will return the ASCII code for backslash ($5C) while Control-K will return the ASCII code for left square bracket ($5B). These are provided to round out the ASCII set which can be generated from the standard Apple keyboard. Backspace, or Control-H, which is generated by the left arrow on the apple keyboard, is changed to ASCII rubout ($7F).

In addition, the key Control-Shift-M is used to force the Apple keyboard to generate lower case. This key toggles the case conversion on and off in sequence. When the lower case switch is on, conversion takes place on all 64 normal ASCII upper case codes.

On the output side of device 1, the routine will monitor the keyboard and hang waiting for another key to be struck if

Control-S was struck on the keyboard.

The device 1 output routines will interpret carriage return, line feed, tab, form feed, bel, backspace and rubout. In addition, they will convert all lower case characters to inverted video and all control characters, that do not have an operational function, into flashing characters.

Device 0 performs using device 1, so in general the above applies.

The line input routine interprets the screen edit functions that are documented in the standard Apple II reference manual. Thus left and right arrow, and the escape sequences perform as usual. Control-X is the line cancel function. The limitations of the standard Apple screen editing features still apply. In addition, the input routine will discard occurrences of line feed (Control-J) and Control-Z.

Device 0 has the ability to input from files by using deferred console execute mode. A flag in the system page controls this function, which will be used in future releases of Apex.

The console handler uses the Apple ROM windowing routines but does not use the vectors at $36 and $38. These vectors are modified by too many Apple programs to be reliable. You may wish to use them to contruct a "variable" device handler, which uses the Control-P command to the Apple ROM Monitor and can talk to any slot.

APEX SYSTEM PAGE

The memory area from $BF00 through $BFFF is used to communicate between programs and Apex itself. This page is called the System page. It´s organization cannot be changed without major modifications to Apex and all programs which run under Apex.

The area between $BF00 and $BF4F is used to store all the global parameters of the particular program in memory at the time. It is saved with, and read from every SAV file.

The area from $BF50 through $BFFF is used to store the system wide global information that does not change with the particular program which happens to be resident.

The funtion of the various globals is covered briefly in this section. Refer to the listings and to other parts of this manual for more detail.

START AND EXIT VECTORS.

Each program has two start and three exit vectors. The start vectors should be jumps to the respective starting points of the program. The start and restart vectors are usually the same.

The exit vectors are the way in which a program must reenter Apex when it runs to completion. A program should not normally exit to the actual Apex entry vectors, instead it should exit to these vectors which will, in turn, enter Apex. This procedure provides a standard way to change the exit mechanism of programs in exceptional cases, such as to disable the trap exit capability in editors, or to implement a batch processing facility under Apex.

VEXIT	Taken when a program runs to normal termination, or when the Control-C key is typed. Normally points to the Apex REENTER point at $BFD0.
VERROR	Taken when a program exits on a fatal error condition. This normally points to the Apex RELOAD entry point at $BFD6.

VABORT Taken when a program is exited by typing the Control-P key. It normally points to the Apex SAVER entry point at $BFD3.

SAVED IMAGE PARAMETERS

The globals DSKMEM and DSKSIZ reflect the values that USRMEM and PROSIZ had when the program was saved. They are set by Apex and should not be changed or set in normal use.

PROGRAM SPACE PARAMETERS

The globals USRMEM and PROSIZ tell Apex where and how large the program to be saved is. USRMEM is the first address, other than the page zero area, to be saved. PROSIZ is the number of pages beyond USRMEM to be saved. These values are setup by the Apex SAVE command when it is used with arguments.

I2L PARAMETERS

These parameters are used for system utilities and are not to be affected by normal programs. The I2LFLG location should contain FALSE=$00 for all normal programs.

BUFFERS

The locations OTBUFD,OTBUFE,INBUFD and INBUFE are used to determine what area of memory the byte I/O to disk files handler (device 3) will use as input and output buffers. These parameters are part of the program space so that each program can select the buffers individually.

These buffers must begin on a page boundary and extend for an integral number of pages.

Note that the KSCAN routine will use the input buffer as a place to put the directory. Thus the input buffer pointers must be valid, and the buffer at least three pages long, whenever KSCAN is used.

The Apex Exec will use the area from $6000 through $63FF for its buffers. The standard BIN file loader will preset the buffers to the memory area from $A000 through $AFFF.

If Apex is entered via the RELOAD or SAVER entry points KSCAN will be used with an input buffer at $A000 to find the system files.

OTHER PROGRAM SPECIFIC GLOBALS

The rerun flag RERUNF is preset by Apex to zero whenever a program is run. The program itself may modify and test this byte at will.

The default input and output extensions are the extensions for input and output files which the program will use. There are two special codings for these:

1) The coding "@@@" means that the program uses the corresponding file but has no suggestion to make about extensions. In this case the OPEN function of Apex will use the extensions from the system default file name as set by the DFILE command.

2) The coding "   ", that is, three space charaters, means that the program does not require this file. Apex will not allow the corresponding file to be opened.

The byte DEFAUL holds the program specific switches. These are the SIZE, BACKUP and KEEP DATE switches. A value of zero in this byte corresponds to all switches off.

The SYBOMB flag tells Apex wether the user program uses the Apex Exec space in memory. It should be TRUE=$FF if the program disturbs any memory above $7000, and to FALSE=$00 otherwise. Strange things will happen if this flag is wrong, particulary if the flag incorrectly holds the special system value of $55. If SYBOMB is set to false, Apex will use the area between $6000 and $6FFF when the executive executes. However, Apex does not require that this area have any particular initial content, so both the user program and Apex may use the space alternately.

The byte USRTOP is intended to hold the (page number + one) of the highest page the current program may use. This byte is irrelevant (but should be set for cleanliness anyway) if the program uses a fixed amount of memory. Some programs, such as editors and assemblers, will use as much memory as possible so they will use this byte to determine how much they are allowed to use. The buffers should always have addresses higher than (or begin at) this page.

USRTOP should be set with due regard for the SYBOMB flag. Both the buffers and USRTOP should never be above $7000 if SYBOMB is FALSE.

INPUT/OUTPUT FILE PARAMETERS

These areas are setup by the OPEN or run operations in Apex. They remain valid while Apex loads and executes a program stored as a SAV file. The program can therefore use them as a simple way to get at the first input and output files.

Apex sets up the starting and ending block numbers, the unit number and the directory number of the files. In addition, Apex will set the corresponding flag byte (INFLG or OTFLG) to 1 if the file area is setup and to 0 if no file was setup.

If the byte I/O to files handler is used, it will use these parameters to access the input and output file as required.

Additional byte I/O input files may be opened by using the SCAN entry point to the byte I/O handler. See the section on using files (pg.10) and the listings.

THE UNIT DRIVER INFORMATION BLOCK

This is an area for use in communicating the multiple parameters required by the unit drivers to those drivers. The system functions KREAD, KWRITE and KSCAN use this block. Refer to the listings for the details.

THE DEVICE HANDLER TABLE

This is a table of eight addresses, one for each possible byte I/O device handler. They contain the address of the start of the handler. They are the ONLY way Apex knows about the handlers and so they are all that you need change when handlers change or move. Access to handlers is via the system function KHAND which will use this address to compute the final address of the routine to be executed (pg. 27).

OTHER SYSTEM DEPENDANT GLOBALS

The flag SYSENF is used by the Apex resident portion to communicate with the non-resident portion, and is not for any other use.

The parameters SYSDEV, SYSBLK and SWPBLK describe the current system unit. A study of the resident code is required for a complete understanding of when these values are valid.

The parameter DEVMSK indicates which Apex units are active on a given system. See elsewhere for the bit mapping used (pg.6).

SYSDAT holds the current system date as a 16 bit integer formed by the equasion:

((YEAR-1976)*16+MONTH)*32+DAY

The byte following SYSDAT is used to indicate the validity of the system date. When valid it should hold the "exclusive or" of the "and" of the two bytes of the system date.

APEX RESIDENT SYSTEM VECTORS

If you look at the listings for the system page you will see that Apex has a set of vectors which provide a single and stable method for other programs (incuding the Apex Exec) to access the code included in the resident portion of Apex. This code should never be accessed by any other means.

APEX ENTRY POINTS

There are three ways in which you may enter Apex from the outside. They all assume that the resident memory area has been correctly setup by the bootup process. The entries are:

REENTER:($BFD0)
This entry point assumes that Apex has been recently run and has all its parameters correctly set. If you enter Apex here, it will decide wether or not it needs to reload, and proceed accordingly. This entry point will assume that the system unit information in the system page is current. The keyboard escape Control-C will re-enter Apex by this vector.

SAVER:($BFD3)
This entry point will reload Apex, but will first save the current memory image in the file SCRATCH.SYS. It is used if the Apex commands SWAP, START or SAVE will subsequently be used. The keyboard escape Control-P will re-enter Apex through this vector.

RELOAD:($BFD6)
This entry point is the cold start entry for Apex. It will reload from the system unit independant of the state of the system page. If Apex is started from a different system unit this entry point should be used.

APEX RESIDENT SYSTEM FUNCTIONS

Some resident system funtions are standard 6502 subroutines which perform some generally useful task. These routines may change any registers and will return with the carry flag set if they failed to perform their task, and clear otherwise.

KHAND   This allows general device independant access to the

	device handlers. See the section on handlers (pg. 25).
KSCAN	This routine provides a simple means for finding files by name on Apex units. Examining the code for this routine will clarify the Apex directory format.
KRESTD	This routine will reset the disk driver routines.
KREAD	This routine is the basic means by which all block oriented reads are done from Apex units. It performs the read in accordance with the parameters placed in the unit driver information block. See the listings.
KWRITE	This is the companion routine to KREAD.

The other vectors provide paths for the basic program execution functions required by the Apex Exec. They would not normally be used by user programs.

COMMANDS

Commands are instructions to the operating system to perform specific operations. A command can be executed by typing the command on the keyboard, followed by a CARRIAGE RETURN. All of the commands in Apex can be abbreviated to two characters. For example DIRECTORY can be abbreviated to DI.

Many commands take arguments. The arguments generally clarify or elaborate upon the command´s action. A SPACE character is used to separate a command from the argument. For example:

```
        [COMMAND] [argument]
```

The following pages outline only the specific aspects of each Apex command. To develop a complete understanding of the commands you must read this entire manual.

RUN

Most operating systems have a "RUN" command. This command is used to execute any program which has been saved as a file. In Apex, the run command is an implied command. Files are executed simply by typing the name of the file.

This allows SAV files to be executed in such a way that they act just like Apex commands. For example, let´s say that you have a program that handles your printer. It is customized so that it does tabs and form feeds and understands all of the idiosyncrasies of your printer. This program could be named PRINT.SAV. Then, whenever you wish to print a file, all you need to do is type word "PRINT" followed by the name of the file you wish to print.

In fact, this concept goes one step further: If a SAV file on your system unit has the same name as a built in Apex command then the file will have priority over the command. Thus you can customize any Apex commands to your own needs without modifying the basic Apex Executive.

A run command normally expects to find the file on the system unit. If the file exists on some other unit, it can be run by prefixing the name with the unit number:

```
        2:PRINT FROG.TXT
```

Executing a program can also have the effect of opening a file for input or output or both. For example:

```
        EDIT FROG.FIL
```

Here Apex will load and execute the file EDIT.SAV. At the same time, it will set up FROG.FIL as the input and output file. When the editor is running, it will read information from the input file and write information to the newly forming output file. More information on opening files is given elsewhere (pg.10, 42).

ZERO

The ZERO command clears the directory of all files, but does not change any other data on the unit. This operation is usually used to set up a new unit or to clear an old one for a different use. The operation will effectively destroy all the file information in directory, so use with caution. Apex will ask you to verify before it will execute the operation. (Note: the directory can usually be restored by using the backup directory operations described below). Example:

```
        ZERO 3
```

MAKE

This command creates a new file. For example:

```
        MAKE FROG
```

or

```
        MAKE 2:FROG.FIL=25
```

or

```
        MAKE FROG.FIL=28,150
```

The first number is the size of the file to be created. The second is the block number to use.

If no size is specified, Apex will create an empty text file with the given name. If a size is specified, the allocated area of the unit will not be cleared and any data left from previous files will be unaffected.

If the starting block is not specified, Apex will start the file at the first empty into which the file fits. If the starting block is given, the file will begin at that block and extend for the specified number of blocks.

The MAKE command does not check for file conflicts in any way. Overlapping files and files with conflicting names can be created. This allows the MAKE command to be used for error recovery or other special purposes. In a situation where the directory has been destroyed and the backup directory is invalid, some files could be restored by setting up dummy files across the unit and examining the files until the lost information is found. A laborious process, but a life saver when the information is important.

MAKE can also be used to set up files in special configuratons to exchange disks with other operating systems.

DIRECTORY

The directory command prints information about the active files contained on a unit. It can be used with no argument, a unit number argument or a file specification argument.

With no file specification argument, the directory command prints a directory of all the files on a unit. With a file specification, information about that specific file will be printed. When fuzzy file names (* and ?) are used, information about each match is printed. As described above, a star (*) will match any file name or extension and a question mark will match any character. Thus

```
        DIR 1:*.P65
```

will print information on any file with the extension "P65" on unit 1 and

```
        DIR XPL?????.*
```

will print information about any file on the task unit whose file name begins with the letters "XPL".

The DIRECTORY command has the ability to accept a switch. The letter "L" is used to select between one directory format and another. With no switches set, a simplified directory is printed. With the "L" switch set, Apex will print out a complete directory that includes file name, size, creation date, and the area of unit allocated. Thus:

```
        DIR 0/L
```

The first line of the directory listing prints the system day and date (not the date on the disk), the unit number and the volume number. The second line contains the title for the particular unit. The last line printed describes the free space on the unit. FREE prints the total empty space on a unit. MAX indicates the largest single free space, which is the largest single file the unit can hold at present.

If the print out of the directory is larger than will fit on the screen, the listing will pause at the end of the screen. Striking any key will print the rest of the directory.

DELETE

This command removes the specified files from the directory. If more than one file is to be deleted, fuzzy file names (* and ?) can be used.

When Apex deletes files from the directory, it will list all of the files that it is going to remove. It will then ask you to verify that these are indeed the files that you wish to remove. "Y" will remove the files "N" will leave them. Under some circumstances, deleted files can be restored using the backup directory described below.

Example:

```
        DELETE 2:FROG.*
```

The default extension for delete operations is BAK. This ensures that you have to type a little more than just a colon to delete the default file. It also allows the command

```
        DELETE *
```

to delete any backup files you have on the task unit.

SAVE

This command is used to save the contents of memory as an executable SAV file. The command takes the name you want it saved as, and optionally, a starting and ending address. If the starting and ending address is not specified, the command will assume that the program area of the system page has the correct values in the locations USRMEM and PROSIZ and will take the information from them. The first 80 bytes of the system page and the area of memory from $0050 through $00FF are automatically included in every saved program. Examples:

```
SAVE 2:FROG=$800,$1000

SAVE PIG
```

The save command requires that Apex be re-enterd at the SAVER entry point immediately prior to the SAVE command. In this way we can be sure that all of the original memory image will be correctly saved, even if part of it was overwritten by the Apex Exec. Apex will take that part of the image to save from the file SCRATCH.SYS, where it will have been saved when Apex was entered at the SAVER entry.

If the memory image to be saved was created by a standard Apex loader, then the loader will have loaded the data and then automatically re-entered Apex through the SAVER entry point. In addition, some loaders (the assembly language binary loader is a notable exception) will also set up the program size and location for you, so that the SAVE command need not have any numeric parameters.

If the memory image was created in some way which leaves you talking to a standard Apex program, or to the Apple ROM monitor, Control-P or Control-Y repectively, will re-enter Apex at the SAVER entry point.

The SAVE command does not affect any system page paramters other than the program location and size. So, unless they were set in some other way, you may wish to set the system related properties of the program using the SET utility after the SAVE.

OPEN

This command opens files for input, output, or both. Files may be opened explicitly, or implicitly using the system defaults.

The OPEN command and a run command funtion in the same way, as far as opening files are concerned. The difference is that the open command will return immediately to the Apex Exec, while a run command executes some other program. Opened files remain open until Apex is re-entered or some other files are opened.

In the general form, the command is followed by two file arguments separated by a less-than sign (the "arrow"). The file name to the right of the arrow is the input file. The file to the left of the arrow is the output file. As in:

```
OPEN OUTPUT<2:INPUT
```

Either the input file or the output file can be omitted, and Apex will only open the file specified. If a single file name, with no arrow, follows the command, the operating system will assume that the file is to be both input and output file. For example:

OPEN B.FIL<A.FIL	Opens "A" for input and "B" for output.
OPEN B.FIL	Opens "B" for both input and output.
OPEN B.FIL<	Opens "B" for output only.
OPEN 0:<2:	Opens default file on units as speciifed.
OPEN <B.FIL	Opens "B" for input only.
OPEN(SPACE)	Opens the default file on the task unit.

The OPEN command is usually used only under special circumstances, since files are normally opened in the process of executing a SAV file. Any SAV file followed by a file name or names will automatically open the specified files. For example:

```
ASM FROG.BIN<FROG.P65
```

In some situations, a unit will be so full that not enough space will be available for the output file. This is usually handled by removing unnecessary files from the unit to make space, possibly followed by a SQUASH operation. When the unit is so tight that no space can be made by deleting files, the operating system allows the output file to be written over the input file.

This is a dangerous operation and should be used with caution. It relies on the fact that the data is buffered in memory buffers. Thus, the input file is usually one buffer load ahead of the output file. However, if the program adds too much information to the output file the output can overwrite the input before the input is read. You must also take care to ensure that the file doesn´t grow to the point that it will no longer fit in the free space. The MAKE command can be used to prevent this by forcing Apex to make the file too large in the first place.

The overwrite feature is enabled by a /R switch in the command line. The switch will operate with both the OPEN command and the implied run command. Examples:

```
OPEN FROG.FIL/R

EDIT FROG.FIL/R
```

INITIALIZE

This command is used to re-write a copy of Apex on a system unit. The SYSTEM.SYS file must already exist on that unit, and be 65 blocks long. The file does not need to contain any valid information at this point however, and so it could have been made by the MAKE command. To write a copy of the currrent Exec into the file, you type INIT followed by the unit you wish to save it on. The system will write itself onto the new unit in the file SYSTEM.SYS. Example:

```
INIT    0
```

Note that INIT also saves the system residual area from $A000 through $AFFF. So if you add handlers in this area you should use INIT to make them permanent.

START

When a program has been saved in the scratch area of the system unit, and has a valid start vector, it can be restarted using the START command. Apex keeps track of when it thinks that there is a valid program in the scratch area. If the system thinks that there is no valid program in the swap area, it will ask you to verify the operation. Typing Y will complete the operation, N will abort it. If you verify a START when the scratch area or start vector is not valid, you can destroy Apex and may have to reboot. This command has no arguments or switches.

SWAP

This command is similiar to START. SWAP brings the program in the swap area into memory but does not execute the program. Instead, the ROM monitor is entered. This is useful for patching programs before they are saved etc. Apex can be re-entered from the ROM monitor with Control-Y. This command has no arguments or switches.

GET

This command is similiar to a run command. The operating system moves a SAV file into memory, but does not begin execution; the ROM monitor is entered instead. The command is used mostly to change or patch programs. The program can be moved into memory using the GET command, then patched, and Apex re-entered with Control-Y. The program can then be re-saved using the SAVE command. Eg:

```
        GET 3:FROG.SAV
```

RENAME

Rename provides a mechanism for changing the name of files. Any file name or extension can be changed to any other name or extension. The operating system checks to see that renaming the file produces no duplicate file conflicts and that the file names both refer to the same unit. For example:

```
        RENAME NEWFILE<OLDFILE.PIG
```

CLOSE

This command unconditionally closes any open file. The file size is left equal to the maximum length of the empty space. Generally, files are closed automatically in normal operations. This command is only used to recover from situations where the program operating on the file has failed to close the file. It allows the data that has been sent to the output file to be recovered.

Note that the file will be too long and will probably not contain an end-of-file mark (Control-Z). The correct file length and end-of-file can usually be restored by editing the file. Use the editor to delete the excess data left in the invalid part of the file.

There are no arguments or switches in the close command.

LIST

This command allows any ASCII text file to be listed on the console device (Apex device 0). The listing can be interrupted with Control-S and aborted with Control-C. Example:

```
        LIST 3:FROG.DAT
```

TITLE

Each unit in Apex can have its own individual title. The title is used to keep track of how each unit is being used. The title can be up to 32 characters long.

The TITLE command is used to set or reset the unit title:

```
        TITLE  MAILING LISTS
```

The command also automatically changes the unit volume number. The volume number is derived from the combination of a random number and the date. The volume number will be used to prevent Apex from writing the wrong directory onto a unit - something that could otherwise happen if you changed disks without informing Apex with the NEW command.

To title a unit other than the task unit, you prefix the new title with the unit number:

```
        TI 4:FOOD FOR THOUGHT
```

DFILE

The DFILE command is used to read or change the system wide file default file specification. If DF is typed without an argument, the system defaults will be printed. If the command is followed by a file name, the system wide file default will be set to this file.

The DF command also prints out the status of the system wide switches PACK, BACKUP and CHECK.

The unit number of the task unit is set by this command and will be whatever unit you assigned the default file to. Eg:

```
        DFILE 6:NEWFIL.TXT
```

or

```
        DFILE 3:
```

or

```
        DFILE   .P65
```

NO and DO

The NO and DO commands are used to set the system wide switches PACK, BACKUP and CHECK (pg. 21). DO will turn any of the switches on, and NO will turn any of the switches off. For example:

```
        NO PACK
```

or

```
        DO BACKUP
```

DATE

Apex maintains a system date which is used to date files as they are created. The date is stored in a part of the directory on each unit, but only the date on the current system unit is used.

The system date should be updated periodically, using the DATE command. When updating the system date, the DATE command must be followed by a carriage return. The system will then prompt you for a new system date. Example:

```
        DATE
        ENTER NEW DATE: 7-4-79
```

The Date is always in the following form: 4-15-79.

The DATE command can also be used to change the date of a file to the current system date. If the DATE command is followed by a file name, the file´s date will be changed to the current system date. Example:

```
        DATE 2:FROG.FIL
```

BDIR

The backup directory is a protection feature of the operating system. Apex maintains two separate copies of the directory on each unit.

A copy of the main directory is made in the backup directory of a unit every time a new output file is opened on that unit.

If the normal directory is destroyed or if a file is accidentally erased, the backup directory, if it is recent enough, can be used to restore the information.

The BDIR command reads the backup directory and displays it for checking purposes. The command alone only reads the backup directory for checking purposes, it does not change the real directory.

A unit number can be appended to the command to refer to a unit other than the task unit:

```
        BDIR 3
```

With the switch /W, the backup directory will be read and written back over the normal directory:

```
        BDIR/W
```

Since the backup directory is only updated when an output file is opened on a unit, certain units, such as system units, may not normally have a valid backup directory. The /B switch, used with the BDIR command, will force the backup directory of a unit to be brought up to date:

```
        BDIR 0/B
```

SYSTEM

This command prints the unit number of the current system unit or changes the system unit to another unit. The new unit must, of course, be a valid system unit. Example:

```
        SYSTEM
```

or

```
        SYSTEM 3
```

SIZE

Apex can deal with units of any size. The size of a unit is maintained in its directory. Sometimes a unit may not be correctly sized for one reason or another. The Size command allows you to check the size of a unit as in:

```
        SIZE 2
```

Or, it can be used to reset or change the size of a unit as in:

```
        SIZE 2=455
```

Apple 5.25" floppy disks are normally 455 blocks long while the single density 8" disks are normally 1001 blocks long.

NEW

This command informs Apex that one or more units have been changed. It is effectively equivalent to typing Control-C while talking to the Apex Exec.

Apex does not read unit directories unless it appears necessary. This means that it is possible to change the unit in a drive without Apex knowing about it. Most often this will just cause a bit of confusion.

Under certain error conditions Apex may try to write the wrong directory onto a unit. If so it will detect a volume number mismatch and abort the operation.

However Apex does not do a volume check on every block written, only on directory changes. So it is possible to damage information on a disk by changing disks at certain critical times and neglecting the NEW command.

APEX STANDARD UTILITIES

As described above, any run file can be executed simply by typing the file name. In this way, programs can be made at act just like commands. Since there are many more desirable functions than can be easily incorporated into the operating system, it makes sense to make some of the commands run file commands. APEX has a number of utilities that are part of the standard operating procedures. The user can also create any utility to fit his needs.

If you have a single drive system all these utilities, with the exception of COPY and DUPDSK, will still work. However you may have to swap disks about somewhat. In general, when a utility asks for a unit number on which to perform a function, you must make sure the correct disk is in place and, if necessary, change the disks **BEFORE** answering the question.

Here is a description of the standard utility programs:

SET

This utility is used to set the program specific portions of the system page in a SAV file. It sets the file related defaults as well as the memory use parameters. See the section on the system page for what these parameters mean.

SET will present you with each parameter in sequence. If you want to change that parameter, enter the new value. If the current value is correct, you simply enter a return.

Since continually SETting a program while debugging is rather a tedious process you may wish to include the code to load the system page values as a part of the actual program source file. However, be clear on what you are loading if you do this, or you may upset the loader or its swapping exit process.

EXCH

This utility is used to copy files from unit to unit on a single drive systems only. It workes only on the system unit. The file to be transferred is specified in the run command line as usual. EXCHange reads a portion of the original file into memory, disks are exchanged, and the portion is written onto the new disk. If the file is large, it may take several exchanges to move the entire file. EXCH keeps track of which disk should be in the drive, and will prompt you to put the correct disk into the drive before each operation.

When the transfer is complete, it will ask you to put in a system unit before it reboots the operating system. Care should be taken that the correct disk is in the drive before each operation. Placing the wrong disk in the drive can result in the destruction of one or more files on that disk.

SQUASH

This utility is used to move all active files to the bottom portion of the unit. This eliminates any small fragments of empty space and creates a single large empty file space.

COPY

This utility copies one or more Apex files from one unit to another. It requires multiple disk drives. The file name given to COPY may be fuzzy if more than one file is to be copied. Copy does not currently pick up the file specification from the run command line. It requires that the file be separately specified in response to the prompt.

BOOTER

This utility is used to update the file RESCOD.SYS to include some modification to the Apex resident memory area and thus make the modification permanent. It will request that you feed it a bootable Apex disk as the master. The bootstrap will be taken from this disk, while the resident code for Apex will be taken from memory.

MAKER

This utility is provided to facilitate the process of setting up a new Apple 5" mini floppy for use by Apex. The disk in question must have been previously formatted with the routine supplied with Apex. See the appropriate setion (pg. 7).

MAKER competely over-writes any previous disk content, and so is used only to create valid Apex disks from formatted empty disks.

In the event that you wish to setup a disk other than a standard Apple disk, you will have to do the operation "manually". The operations which MAKER does for you are equivalent to the following steps (in sequence):

```
        TITLE the disk.
        SIZE the disk.
        ZERO the disk.
        Set the date.
        Set the default file.
        MAKE the file RESCOD.SYS=9,17
        MAKE the file SCRATCH.SYS=65
        MAKE the file SYSTEM.SYS=65
        Use INIT to write the system onto the disk.
        Use BOOTER to write the resident code onto the disk.
```

LOAD

This program is the standard Apex mechanism for loading the binary files produced by the assembler. It will accept a list of input files, which will all be loaded in sequence:

```
        LOAD FROG,DOG,PIG
```

After the load is complete the loader will re-enter Apex at the SAVER entry point. You can then use SAVE, SWAP, or START commands as decribed earlier.

Unlike high level language loaders, this loader presets the system page parameters to safe values, but does not try to divine the "correct" setting for them. You must set them yourself, either as a part of the load, or by using the appropriate Apex commands and the SET utility.

The binary loader overlays itself onto the system residual area. Thus it cannot be used to load this area directly. It also follows that it must force the SYSBOMB flag to TRUE, since the Exec is indeed bombed. If the program that was loaded does not need this flag TRUE then the flag may be reset to FALSE after the program has been made into a SAV file.

More detail on the loader can be found in the assembler manual.

DUPDSK

This simple program can be used to copy entire units from one to another. It operates on a block by block basis without any reference to content. DUPDSK is used when a unit oriented copy is required, such as if you have a mix of different drives, or when your units do not correspond to separate disks.

If you simply want to duplicate an Apple mini-floppy, the Apple supplied dual drive disk copy is simpler because it also transfers the formatting.

COMMAND SUMMARY

ZERO		Clears directory.
MAKE		Creates a file.
DIRECTORY		Prints a directory.
	/L	Prints the long form of the directory.
DELETE		Removes specified files from the directory.
SAVE		Saves specified file from swap area.
OPEN		Opens specified files for input and/or output.
	/R	Sets output file to overwrite input file.
INITIALIZE		Re-writes the operating system on the unit.
START		Reloads and starts the saved memory image.
SWAP		Reloads the saved image and enters the monitor.
GET		Loads a named file and enters the monitor.
RENAME		Renames a file in the directory.
CLOSE		Unconditionally closes any open output file.
LIST		Prints the named file on the console.
TITLE		Changes the unit title.
DFILE		Shows or sets the system wide default file name.
DATE		Changes the system date or a dates a file.
BDIR		Shows the backup directory of a unit.
	/W	Overwrites the main directory with backup.
	/B	Forces a backup directory to be updated.
DO		Enables the specified system wide switch.
NO		Disables the specified system wide switch.

SYSTEM	Shows or changes the system unit number.
SIZE	Shows or changes the size of a unit.
NEW	Ensures that Apex knows that you changed disks.

APEX MEMORY MAP

```
C000- -\                         \         \
       |APEX SYSTEM PAGE         |         |
BF00- -+                         |         |
       |RWTS DISK DRIVERS        |         |BOOT
B800- -+                         |         |BLOCKS
       |COMPAGE                  |         |
B700- -+                         |SYSTEM  -+
       |CONSOLE HANDLER          |RESIDENT |
B480- -+                         |         |
       |                         |         |RESCOD.SYS
B400- -|SYSTEM FUNCTIONS         |         |
       |                         |         |
B000- -+                        -+        -+
       |BYTE I/O HANDLER         |         |
AD00- -+                         |         |
AC00-  |OTHER DEVICE             |         |
AB00-  |HANDLERS                 |SYSTEM   |
AA00- -+                         |RESIDUAL |SYSTEM.SYS
       |                         |         |
A800-  |RUNTIME SYSTEM           |         |
       |                         |         |
A000- -+                        -+         |     \
       |                         |         |     |
9000-  |                         |         |     |
       |APEX EXEC                |         |     |
8000-  |                         |SYSTEM   |     |
       |                         |SCRATCH  |     |
7000- -+                         |FILE     /     |
       |EXEC SCRATCH SPACE       |               |
6400- -+                         |               |
       |EXEC BUFFER SPACE        |               |
6000- -+                        -/               |
       |                                         |
5000-  |                                         |USER
       |                                         |PROGRAM
4000-  |                                         |SPACE
       |FREE SPACE                               |
3000-  |                                         |
       |                                         |
2000-  |                                         |
       |                                         |
1000-  |                                         |
       |                                         |
0100- -+                                         |
       |USER PAGE 0                              |
0050- -+                                         /
       |APPLE ROM SPACE
0020- -+
       |VERY TEMPORARY
0000- -/
```