SK★DOS™

68K

USER'S MANUAL

SOFTWARE SYSTEMS CORPORATION

P.O. BOX 209 · MT. KISCO, NY 10549 · 914/241-0287

SK*DOS®
68K User's Manual

Peter A. Stark

Some earlier versions of 6809 SK*DOS were formerly known as STAR-DOS.

This is revision 1.12 of the manual, last revised on June 11, 1988.

CONTENTS

BEFORE STARTING

In general, it is important that you develop good habits when using any floppy disk system. It is important that you make frequent backup disks, since it is very easy to lose a file, or even the data on an entire disk, due to a slippery finger or careless mistake.

Since a Disk Operating System (DOS) is an extremely powerful program which allows you to access the disk on a most elementary basis, exercising caution and making frequent backups is especially important.

If possible, you should make a backup of the SK*DOS system disk before doing anything else. If your SK*DOS was supplied to you already configured for the disk controller and other hardware you now have, then making such a backup is easy; if your hardware is different from what SK*DOS needs then it is not.

Assuming that your hardware is capable of running this version of SK*DOS as is, here is what to do to get a backup.

(1) Make sure the original SK*DOS disk is write-protected. On a five-inch disk you should place the write-protect tab over the slot on one edge of the disk; on an eight-inch disk you should remove the write-protect tab from the disk.

(2) Place the SK*DOS disk into drive 0, close the door, and boot it.

(3) After SK*DOS is booted and you get the prompt, use the FORMAT command to initialize a blank disk in drive 1. See Appendix G for instructions on using FORMAT. Format the disk in double density, including track 0.

(4) Once a new disk is formatted, use the BACKUP command to copy the SK*DOS system disk onto the blank disk. (See Appendix G for instructions on BACKUP.)

(5) Make several such backup disks, and put the original SK*DOS disk away into a safe place.

In any case, it might be a good idea to read this entire manual before proceeding.

PLEASE NOTE: Several years of work and effort have gone into writing and improving SK*DOS. It pains us to see this work taken lightly by individuals who give away or sell bootleg copies of this software or this manual. Please don't do it.

0. ABOUT THIS VERSION

As you know, SK*DOS/68K is a relatively new product. Many of our users have suggested changes and additions to SK*DOS which we have been happy to receive and incorporate. In the process, we have also tried to keep the manual current and complete at all times. Yet it is still quite possible that it lacks information on features in your version, or that new features will be added to SK*DOS after you receive your program and manual. You should contact Star-K Software Systems Corp. periodically (as described on the User Registration form) to check on possible updates to your software. If possible, use the Star-K computer BBS to contact us for support and further information.

The standard 68K SK*DOS disk uses double density on all tracks. Note that this is different from 6809 SK*DOS disks which may be single or double density, but whose track 0 is always single density. SK*DOS/68K will read and write either single or double density disks, so interchanging disks with 6809 systems is no problem. Note, however, that some 68K systems cannot boot from a single density disk, although they can use single density once booted. Hence you should always format disks in double density if you intend to use them on your 68K system, and single density if you intend to use them on a 6809 system.

To boot SK*DOS use the FD, WD, WA, or WB commands in the HUMBUG monitor (see the HUMBUG manual) or follow the prompts in your boot ROM.

1. INTRODUCTION

The Disk Operating System, or DOS for short, is a program which acts as a file manager for a disk. The DOS acts as a buffer between the disk hardware, and the software which uses that disk. Its primary function is to maintain a disk directory on each disk, fetch program or data files from the disk as needed, and store programs or data back on the disk.

SK*DOS consists of three major parts:

(1) The Command Processor System or CPS, which is the major interface to the user. When SK*DOS is active, the CPS monitors the keyboard and waits for user commands. At that time, you can load and execute programs from the disk and do certain other functions. In addition, the CPS has a number of routines which can be used by other programs to simplify input and output for the terminal.

(2) The File Control System or FCS is the interface for programs running under SK*DOS. The FCS does the actual work of managing the contents of the disk. It has various routines which can be called by user programs for managing the disk contents.

(3) Memory- and disk-resident commands provide additional functions which work in conjunction with the CPS and FCS to provide an easy way of maintaining the disk.

In addition to the various commands supplied with SK*DOS, there are other programs available from other vendors which are designed to also work with SK*DOS. Furthermore, SK*DOS/68K floppy disks are compatible with those used by 6809 SK*DOS and Flex (a trademark of Technical Systems Consultants), so text and data files may be brought over from 6809 systems and used on your 68K system as well.

The interface between SK*DOS and user programs is similar to that standardized over a number of years on earlier 6809 systems, and is fully documented in this manual.

Since a DOS must be customized to run on a particular system, Star-K Software Systems Corporation, in conjunction with manufacturers who license SK*DOS, provides several different versions. Depending on the hardware configuration you specified with your order, you may already have received a version which is customized for your hardware, or you may have received a generic version which must still be adapted to your particular hardware. In general, you will have one of the following two files on your disk:

SK*DOS.SYS is a bootable version (which is started with the appropriate boot command of your monitor) and which is already configured for your hardware.

SK*DOS.COR is a generic version which lacks all console terminal and disk driver interfacing. In order to run this version, you will have to provide your own interface routines. An SK*DOS Configuration Manual, available from Star-K Software Systems Corp., describes the process of configuring SK*DOS for other hardware systems, and shows sample interfacing routines which can be used as a guide.

2. FOR THE IMPATIENT ...

If you're anything like us, you want to try out a new program even before reading the manual, just to make sure 'it works'. This is difficult to do with something as powerful as a DOS, but just to show you a bit of what SK*DOS can do, this section shows you how to bring the system up and run it. (This is only possible at this time if you have received a version of SK*DOS which is already configured for your specific hardware.)

After you finish trying it out, we suggest you put it away and go back to reading this manual.

To start, make sure that your disk is write-protected and place it into drive 0. Read Section 0, About This Version, for information regarding how to boot SK*DOS on your system.

The computer will load the program and respond with

```
          WELCOME TO
          SK*DOS/68K
(C) 1986, 1988 BY PETER A. STARK
STAR-K SOFTWARE SYSTEMS CORP.

ENTER TODAY'S DATE (MM,DD,YY):
```

Respond with the date, using one or two digits for the month and day, and two digits for the year, as in 9,26,82, and hit the ENTER key. (If your system has a calendar clock and it has been set to the correct date and time, SK*DOS may take its date from the clock instead of asking you to enter it.) You will now get the prompt

```
SK*DOS:
```

SK*DOS is now running, and awaiting your further command. (You are looking at just the tip of SK*DOS - the part which is visible to the user. There is much more to SK*DOS than this, but this is the only part which you can see and experiment with without doing a bit more reading.)

You can now type in a variety of commands. Some commands will be immediately recognized by SK*DOS and acted upon; these are called 'memory resident' commands. Others are not recognized, and so SK*DOS will try to find them on the disk; these are called 'disk resident' commands.

To try a memory resident command, type the word

```
XEQ
```

and hit ENTER. XEQ tells SK*DOS to execute the last program loaded in by SK*DOS. Of course, we haven't yet used SK*DOS to load anything, and so we get an error message which reads

```
ERROR 28 - MISSING TRANSFER ADDRESS
```

SK*DOS error codes are explained in Appendix E; in this case there is no transfer address so SK*DOS does not know what to execute.

Let us next execute a disk-resident command:

```
ACAT 0
```

The ACAT command prints an alphabetized catalog of the disk or current directory. (ACAT or ACAT 0 means drive 0, ACAT 1 means drive 1, and so on). In response to the ACAT 0 command, SK*DOS loads the ACAT.COM program from the disk and executes it, displaying a catalog of the disk in drive 0. (To halt the listing, just press the ESC or escape key. Pressing ESC again will continue the catalog listing, or pressing CR or RETURN will return to the SK*DOS prompt.)

Once the ACAT command is finished, you may repeat the entire command by pressing control-A. The control-A displays the entire previous command line (ACAT 0) as if you had typed it again. At this point you may simply perform that command by pressing CR, or may backspace and change all or part of the line. For example, you may backspace to the 0 and replace it with a 1 (assuming you have a drive 1), thereby changing the command to ACAT 1. Pressing the CR now would result in a catalog display of the disk in drive 1.

Another way of repeating a command is by using the XEQ command. XEQ tells SK*DOS to execute the last program loaded by SK*DOS. In this case, the last disk-resident command loaded is ACAT, so XEQ repeats the ACAT command.

There are two interesting points to note:

1. Running ACAT again by typing XEQ is faster than typing ACAT again, because typing ACAT loads the program from disk and then executes it, whereas XEQ merely restarts the ACAT program without loading it again.

2. Pressing control-A repeats an entire previous command line, including any arguments typed as part of that line (for example, the 0 in ACAT 0 is the argument and specifies the drive number.) XEQ restarts a program but does not supply any arguments; these must be entered after the XEQ (as in XEQ 0).

It is now time to return to reading the rest of this manual, so type the memory resident command

```
MON
```

This command exits SK*DOS and returns to the monitor (assuming that your computer has one.)

3. FILE SPECIFICATIONS

The term File Specification or just file-spec refers to the four items required to completely specify a file on a disk: the drive number, a directory name, the file name, and an extension.

The file-spec usually looks something like this:

```
0.A/FILENAME.EXT
```

In this example,

0 is the drive number. It is separated from the rest of the file-spec by a period, and is usually a number between 0 and 9, inclusive. The drive number (and its period) may not always appear; if it is missing, then SK*DOS uses one of two default values (which may both be the same): it uses the *system drive* default value when loading commands from a drive, and it uses the *work drive* default value for most everything else. The drive number may either precede the rest of the file-spec, as above, or may follow it.

A is the directory name. An SK*DOS disk has a main directory, sometimes called a *root* directory, and may have up to 26 smaller directories called *subdirectories*. The subdirectories are called A/ through Z/, while the root directory is either called / or is not specified at all. The directory name (and its slash) may not always appear; if it is missing, then SK*DOS uses either the system or work drive default directory.

FILENAME in the above example is a one- to eight-character long word which usually is chosen to identify the contents of the file. It must begin with a letter, and the remaining characters may be either letters, numbers, hyphens, asterisks ("stars"), or the underscore. The file name is the only absolutely required part of the file-spec in every instance.

EXT in the above example is a one- to three-character abbreviation which usually identifies the *type* of file. It is separated from the name by a period (and may also be followed by another period and the drive number). Like file-names, extensions must begin with a letter, and the remaining characters may be either letters, numbers, hyphens, asterisks ("stars"), or the underscore. Extensions are not always required if they are obvious, since SK*DOS programs default to certain extensions for certain commands.

Each of the four parts of the file-spec needs some further clarifying details:

1. SK*DOS differentiates between *physical drive numbers* and *logical drive numbers*. The physical drive number is the wired-in number determined by the hardware controller and the actual disk drive. For example, a given floppy drive may have its DS0 (drive select 0) jumper installed, which gives that drive the physical number of drive 0. This same drive also has a logical drive number, which may be different from the physical number. This assignment is handled by the DRIVE command, which may assign any of the ten logical drive numbers to any physical drive. The file-spec uses logical drive numbers, not physical drive numbers.

2. Floppy disks usually have only a root directory; most people use subdirectories only to split a large hard disk into smaller, more easily manageable sections. Each subdirectory might then be used for a specific type of file. (Incidentally, if you change the variable FNCASE to allow both upper and lower case file specs, you can then have a total of 52 subdirectories on a disk, called A through Z and a through z.)

3. There are several file names which are reserved for SK*DOS's own internal use. These fall into two categories:

a. The names DIR, GET, GETX, MON, RESET, SAVE, TRACE***, and XEQ are used for internal 'memory-resident' commands. While these names may also be used for file names, any command files bearing these names will not be called since SK*DOS will use its internal commands instead.

b. The names CONS, PRTR, and NULL (and possibly a few others of your own choosing) apply to I/O devices rather than files. These names can also be used for file names, but in certain cases SK*DOS will interpret them as applying to a device rather than a file.

4. While files can have any valid extension, by convention certain kinds of files tend to have specified extensions. These extensions are listed in Appendix F. Most programs and commands default to specific extensions unless another extension is specified. For example, the EDLIN command is used to edit text files, and always assumes that its text files use a .TXT extension if none is given. The GET command, on the other hand, is always used to load binary files into memory, and so it automatically assumes a .BIN extension if none is given. You may provide an extension when you use these commands, but it is usually not necessary.

Unlike default extensions, which are built into commands and programs, default drive numbers and directories can be specified by the user by using the SYSTEM and WORK commands. When SK*DOS is first booted, it defaults everything to the root directory of drive 0. You may keep these defaults, but specify different drive numbers or directory names just when you need them, or you may set up a default drive or directory different from the root directory of drive 0, which will then automatically be chosen for most commands - unless you specify otherwise.

Here is a simple example. Suppose you wish to use the LIST.COM command to display a file called FILE.TXT. Let's assume that LIST is in directory C of drive 0, while FILE.TXT is in directory X of drive 3. One way to do this would be to give the command

```
SK*DOS: 0.C/LIST.COM 3.X/FILE.TXT
```

But since SK*DOS automatically defaults to .COM extensions for commands, while LIST defaults to .TXT extensions for text files, the command could have been shortened to

```
SK*DOS: 0.C/LIST 3.X/FILE
```

If you expect to use a lot of commands from drive 0 directory C, and perhaps a lot of text files from drive 3 directory X, then you could specify those as default values with

```
SK*DOS: SYSTEM 0.C/
SK*DOS: WORK 3.X/
```

These two commands would tell SK*DOS to use 0 and C/ for loading its system commands, and to use 3 and X/ as the work drive and directory. Once you do this, then the command to list the file would simply be

```
SK*DOS: LIST FILE
```

Subdirectories are generally used to organize the contents of a hard disk. For example, you might put all assembler source files into the A/ subdirectory, put your Basic programs into B/, put C programs into C/, and so on.

Although it wastes space, it is possible to put separate copies of a file into more than one directory. For example, to place a copy of the above 3.X/FILE.TXT into subdirectory Y, you could simply use the COPY command as follows:

```
SK*DOS: COPY 3.X/FILE.TXT 3.Y/FILE.TXT
```

To *move* a file from one directory into another, you use the RENAME command. Thus the command

```
SK*DOS: RENAME 3.X/FILE.TXT 3.Y/FILE.TXT
```

leaves only one file on the disk, but moves it from X/ to Y/. This is different from the COPY command, which would leave two copies of the same file on the disk, one in X/ and another in Y/. Note that in this case, changing one copy does not change the other.

4. SK*DOS OVERVIEW

The SK*DOS package consists of essentially two parts:

1. The SK*DOS program itself.

2. A set of disk-resident commands such as CAT and LIST. The disk-resident commands are described later; this section deals with the SK*DOS program itself.

The SK*DOS program in turn consists of three parts:

1. The File Control System (or FCS for short) which maintains the disk directory and in general is responsible for managing the disk contents. The FCS has various routines which may be called by other programs; some of these routines actually handle the disk and its files, while other routines may handle peripheral functions (such as printing out strings, or converting numbers to and from decimal). These routines are used by the FCS, but they are documented in this manual and may also be used by application programs.

2. The Command Processor System (or CPS for short) which acts as an interface between a user and the FCS. When SK*DOS is first loaded and executed, the CPS prints the SK*DOS: prompt and awaits further instructions. These instructions may or may not involve the FCS.

3. The Basic Input/Output System (BIOS) which adapts SK*DOS to run on a particular hardware system. The BIOS contains the software which interfaces with the keyboard, display, printer, and disk drives.

The FCS and CPS parts of SK*DOS are the same on all systems; the BIOS portion must be tailored for each different computer. In some cases, the BIOS is provided either by Star-K or by a hardware manufacturer; in other cases, you may have purchased a more generic version of SK*DOS and will have to provide your own BIOS if your hardware is different from those systems currently supported.

Depending on the version, SK*DOS (with all its hardware-dependent routines) occupies approximately 16K to 24K of RAM. The exact location of SK*DOS varies from system to system, but in most systems it begins at $1000. Once booted, however, it is not necessary to know exactly where SK*DOS is located in your memory, since SK*DOS is called by application programs through 'exception vectors'. Hence this manual describes the FCS and CPS portions of SK*DOS from the point of view of a user or applications programmer. BIOS information, needed only by systems programmers implementing SK*DOS on a new system, is provided in a separate (optional) Configuration Manual.

5. THE COMMAND PROCESSOR SYSTEM

When SK*DOS is first loaded and started, it responds with the SK*DOS: prompt and waits for a command to be processed. At this point you may enter either a Memory Resident Command, a Disk Resident Command, or a Batch Command. Depending on the command, sometimes several commands may be entered on one line, separated by colons. Commands may consist of more than one part and may be up to 127 characters long, if necessary. The various parts of a command may be separated by either spaces or commas, and the command should be followed by a RETURN (which may be labelled ENTER or CR on some keyboards.)

Commands typed into the command processor (and other input which may be entered in user programs) are stored in a line buffer (called LINBUF). If you type a control-A character while inputting a command or other input into the line buffer, SK*DOS will add the remaining contents of the line buffer to the input you have just typed and display it on your terminal. This is useful for repeating a command in its entirety, or repeating just parts of it. For example, suppose you have just completed the command

```
COPY 0.NAME1.TXT 1.NAME2.OLD
```

and then, upon completion of that command, you type a control-A. SK*DOS will display the entire previous line again and position the cursor after the D in OLD. If you then press the RETURN key, you will perform the command again. Alternatively, you may backspace and change any part of the command. For example, if you backspaced to the beginning of NAME2 and typed in a new name such as NAME3, then the line would read

```
COPY 0.NAME1.TXT 1.NAME3
```

At this point you could either press RETURN and execute the line as is, or again press control-A; the latter would complete the line by adding the .OLD at the end.

Whenever you enter a command line, SK*DOS will first check whether you have typed the name of a memory-resident command. If so, the command is immediately executed.

If no memory-resident command exists by that name, SK*DOS will try to find the command on the disk and execute it. Disk resident commands are program files which have a .COM extension.

If no such .COM file is found, SK*DOS makes one more try - to find a Batch File. A batch file is a text file (having a .BAT extension) which itself contains one or more other memory- or disk-resident commands which should also be executed.

For example, suppose a batch file called TWODIR.BAT has the following two text lines:

```
DIR 0
DIR 1
```

Entering the command TWODIR would then display the disk directories of both drives 0 and 1.

If a command is given which is none of the above then SK*DOS will print a 'file not found' error message.

Note that a .BAT file should never have the same name as a .COM file, since SK*DOS will find and execute the .COM file first, and never execute the .BAT file at all.

6. MEMORY RESIDENT COMMANDS

SK*DOS recognizes a number of Memory Resident Commands. These are commands which are integral parts of SK*DOS, and are in memory at all times. They are therefore called by their names, and do not get either an extension or drive number.

Memory-resident commands include the following; more detailed descriptions are provided in Appendix G:

DIR	Display the contents (directory) of a disk
GET	Load a binary file from disk into memory
GETX	Load a binary file from disk into memory
MON	Exit SK*DOS and return to a ROM monitor
RESET	Exit SK*DOS and return to a ROM monitor
SAVE	Save contents of memory to a binary disk file
TRACE***	Allow command tracing; see TRACENAB in Appendix G.
XEQ	Execute the last file loaded from disk

There are some differences between GET and GETX, and between MON and RESET:

GET and GETX both load a binary file from disk into memory, but GET checks the loading addresses and will return an error message if the file would load outside the normal user-accessible memory (as defined by two variables called OFFSET and MEMEND.) GETX does not check such loading addresses.

MON and RESET both exit SK*DOS and return to a ROM monitor, but in some systems MON will re-enter the monitor without doing a complete system initialization; whereas RESET will completely initialize the system. This distinction is important because RESET will initialize all exception vectors to the ROM monitor's values, whereas MON may not. (In some systems, RESET may not be operational.)

Although DIR will display the contents of a disk, you may prefer to use some of the other disk-resident commands such as CAT, ACAT, SCAT, or TCAT, which provide more information than DIR.

Although, strictly speaking, Control-A is not a command, this may be a convenient place to discuss it. Pressing a control-A while typing any command will repeat the remaining portion of any previous command, display it on the screen, and ready it for execution. The control-A is a powerful feature, but it can also be misused if entered past the end of the last previous command.

7. DISK RESIDENT COMMANDS

SK*DOS is supplied with a variety of disk-resident commands which are described in Appendix G. In addition, it is relatively easy to write your own command files and store them on the disk for later execution.

Disk resident commands are supplied as binary, machine language files with .COM extensions. They are executed simply by typing their names. For example, the CAT.COM command may be executed just by typing CAT after the SK*DOS: prompt. This is equivalent to typing the sequence

```
SK*DOS: GET CAT.COM

SK*DOS: XEQ
```

since any unknown word (other than one recognized by SK*DOS as a memory resident command) is interpreted as calling a disk command and executing it. A .COM extension is assumed, and the program is automatically executed as soon as it is loaded.

Since SK*DOS automatically searches the disk for commands, it is possible for users to write their own Disk Resident Commands. Arguments to be used by the command can be entered on the same line as the command name, and then processed by the command with the aid of SK*DOS routines such as NEXTCH (get the next character from the Line Buffer). The listing of the LIST command, later in this manual, will show how additional commands can be written.

The disk-resident commands supplied with SK*DOS are described in Appendix G.

8. THE FILE CONTROL SYSTEM (FCS)

The FCS is the heart of SK*DOS, since it is responsible for reading, writing, and locating files on the disk. Although the FCS system is working, it is invisible when you run some of the disk resident commands such as BUILD or LIST. It is, however, heavily used by all programs which run under SK*DOS. The following explanation will assume that you have the knowledge and need to examine the operation of SK*DOS on this detailed level.

When reading a file, the FCS looks up the file location in the system portion of the disk (track number 0 on the disk) and then goes to read it. When writing a file to the disk, the FCS uses the system track to assign space to the file; when the file is written, the FCS updates the system track so that the file can later be found.

Fortunately, though this process is rather complex in any disk system, the user need not be concerned with how it is done, or where on the disk a given file is located. The SK*DOS FCS does all this automatically; the user need only give the FCS a file name and a command as to what to do. This is done by setting up a File Control Block or FCB for each file that is to be opened; a given program may use as many FCBs as desired. The FCB contains the file-spec, assorted flags and variables which are used by the FCS to keep track of the file, and also the data read from, or about to be written to, a single sector on the disk.

For example, to access the disk through the FCS to read text from a disk file, the sequence of operations would be something like this:

1. Set up a File Control Block with a DS instruction.

2. Point the A4 register to the FCB, and call the SK*DOS FCS system to input a file name.

3. Call SK*DOS again to assign a default extension, if needed.

4. Call SK*DOS a third time to open the file.

5. Call SK*DOS to read a byte from the file and process it.

6. Repeat step 5 as long as needed, then

7. Call SK*DOS to close the file.

All of these operations use the FCB as a buffer, both to hold the contents of an entire sector of data read from or written to the disk, as well as to keep track of the file name and location, and other pertinent data.

The FCB is discussed in the next Chapter.

9. THE FILE CONTROL BLOCK

The File Control Block, or FCB for short, is used for all communications between the File Control System (FCS) and user programs. One FCB is required for each file that is opened by a program, although that FCB can be reused again if a file is closed (thereby releasing the FCB) and another file is opened with the same FCB location.

The FCB is an area of memory 352 or 608 bytes long which must start on an even address. (Although only 352-byte FCBs are used at this time, users should set aside 608 bytes for each FCB so as to be compatible with future versions of SK*DOS.)

SK*DOS maintains several such FCBs for its internal use. One of these is called the User FCB, or USRFCB. It is available for use by other programs. Although it is used by SK*DOS, this is done in a way which does not prevent its use by those programs which also require a FCB.

The FCB consists of 352 (or 608) bytes. Of these, the first 96 bytes are used for storage of various file parameters, while the remaining 256 (or 512) bytes hold the data for one sector of disk data. During a disk read operation, these bytes hold the contents of the last sector previously read; during a write operation, these bytes generally hold the contents of the next sector to be written.

Not all of the first 96 bytes are used; the following descriptions cover those bytes that are used in SK*DOS.

Byte No. 0. Reserved

This byte is reserved for future expansion.

Byte No. 1. Error code (see Appendix E)

After the FCS is finished doing an FCB operation, it returns to the user program. If no error is found, then the Z bit in the condition code register is a 1 and the content of this byte is irrelevant. But if an error is found, then the Z bit is a 0 and this byte contains an error code. The status of the Z bit should be tested directly after returning from SK*DOS with a BNE (to error routine) or BEQ (to normal continuation) instruction. The content of this error byte is also stored in ERRTYP.

Byte No. 2. Read/Write/Update status

This byte indicates whether the file is open for reading, writing, or random file updating. SK*DOS checks the byte prior to reading or writing to make sure that the file is open for the appropriate operation. The values of this byte for an open file are as follows:
1 = open for sequential reading
2 = open for sequential writing
3 = open for updating, but no changes have been made to current sector
83 = open for updating, and changes have been made to current sector (this is hexadecimal 83)

Byte No. 3. Logical drive number (0 through 9)

This byte contains the number of the drive being used for this file control block. The drive number will normally be a number from 0 through 9, but when opening a file for reading or writing it may also be specified as $FF, in which case SK*DOS will search your drives, beginning with drive 0, for a drive where the requested operation can be completed. Then it will place the correct drive number into this byte and open the file. Since many disk drivers do not provide a way of determining whether a 5" floppy disk is ready or not, SK*DOS may "hang up" if there is no disk in a drive being searched, although the DRIVE command can be used to define non-existent drive numbers. (See also Chapter 13 for I/O redirection and its use with the logical drive number.)

Bytes No. 4-11. File name (8 bytes)

These eight bytes contain the name of the file being used with this FCB. The first character of the name (always in byte 4) must be a letter, and the remaining ones may be either letters, digits, hyphens, or underlines. If the name is shorter than 8 characters, then the remaining bytes must be filled with zeroes.

Bytes No. 12-14. Extension (3 bytes)

These three bytes contain the extension that goes with the name in bytes 4 through 11. The extension obeys the same rules as the name described above.

Byte No. 15. File attributes

This byte defines the type of user access permitted to this file. The bits are used as follows:
Bits 0-3 - reserved for future use (leave at 0)
Bit 4 - will not be listed by CAT utility
Bit 5 - Reading not permitted
Bit 6 - Deletion not permitted
Bit 7 - Writing not permitted

Bytes No. 16-17. Reserved

These bytes are reserved for future expansion.

Byte No. 18. First track of file
Byte No. 19. First sector of file

These two bytes point to the first sector of the file.

Bytes No. 20-21. Reserved

These bytes are reserved for future expansion.

Byte No. 22. Last track of file
Byte No. 23. Last sector of file

These two bytes point to the last sector of the file.

Bytes No. 24-25. Number of sectors in the file

These two bytes indicate the size of the file in sectors.

Byte No. 26. Random file indicator

This byte indicates whether the current file is a sequential file or a random file. A zero in this byte indicates a sequential file, a nonzero indicates a random file. (See Chapter 13 for further information on random files.)

Byte No. 27. Time or sequence number

This byte normally contains either the file creation time (encoded as a one-byte number), or a sequence number. Sequence numbers are sequential numbers, beginning with 1 when the system is first booted. Sequence numbers indicate the order in which files are written on any particular day. (See INTIME in Chapter 13 for further information.)

Byte No. 28. Month of file creation (1 through 12)
Byte No. 29. Day of file creation (1 through 31)
Byte No. 30. Year of file creation (last two digits)

These three bytes hold the date when the file was created. All three bytes are in binary, but only the last two decimal digits of the year are stored. That is, in 1984 byte 30 stores a decimal 84, or a hexadecimal 54.

Byte No. 31. Reserved

This byte is reserved for use by SK*DOS.

Bytes No. 32-33. Reserved

These bytes are reserved for future expansion.

Byte No. 34. Current track number
Byte No. 35. Current sector number

These two bytes contain the track and sector number of the sector currently in the FCB. If the file is being read, then they indicate where the data currently in the FCB came from; if the file is being written, then they indicate where this data will go.

Bytes No. 36-46. Temporary name buffer 1

These eleven bytes are used to temporarily hold a file name and extension while the file is being renamed or deleted.

Byte No. 47. Reserved

This byte is reserved for use by SK*DOS.

Byte No. 48. Reserved

This byte is reserved for future expansion.

Byte No. 49. Sequential data pointer to next byte (4 through 255)

On all sequential read or write operations, this byte points to the next byte to be read or written into the sector buffer portion of the FCB. The pointer is a 4 for the first byte, or 255 for the last byte. SK*DOS changes this byte automatically; users will not normally touch it.

Byte No. 50. Reserved

This byte is reserved for future expansion.

Byte No. 51. Random data pointer to next byte (4 through 255)

On all random read or write operations, this byte points to the next byte to be read or written into the sector buffer portion of the FCB. The pointer is a 4 for the first byte, or 255 for the last byte. Unlike the sequential data pointer (byte 49), this byte is not changed by SK*DOS, but is to be set by user programs instead. (See Chapter 13 for further information on random files.)

Bytes No. 52-62. Temporary name buffer 2

These eleven bytes hold the new name and extension of a file being renamed. The new name should be stored into these bytes prior to calling the rename function of the FCS, using the same rules as apply to bytes 4 through 11 above. (These bytes overlap with some of the bytes below, but there is no conflict as they are used at different times.)

Byte No. 58. Column Counter (for Basic)

This byte is used only by Basic to keep track of the current output column.

Byte No. 59. Space compression indicator

This byte indicates whether space compression is being done in reading or writing the current file. Values of 0 through 127 ($00 through $7F) indicate that space compression is being done, and the actual value represents the number of spaces currently being compressed. A value of 255 ($FF) indicates that no space compression is being done. SK*DOS initializes this byte to 00 upon opening a file; it is up to the user to change it to $FF (after opening the file but before reading or writing) if space compression is not desired.

Byte No. 60. Number of sectors per track

This byte contains the number of sectors per track during random file operations. (See Chapter 13 for further information on random files.)

Byte No. 63. Reserved

This byte is reserved for use by SK*DOS.

Bytes No. 64-67. Reserved

These bytes are reserved for future expansion.

Bytes No. 68-71. Next FCB pointer

These four bytes point to the next FCB, if any, which was opened after this one (or, more exactly, they point to the corresponding bytes of the next FCB, not to the beginning of that FCB). This information is used by SK*DOS to keep a list of all FCBs currently in use, so that they can be closed if an FCSCLS operation is requested. If this is the last FCB in the chain, then these bytes contain zeroes.

Byte No. 72. Physical Drive Number

Using the DRIVE command, SK*DOS allows you to reassign logical drive numbers to different physical drives; this byte contains the physical drive number actually used by the hardware in reading or writing a sector or file.

Byte No. 73. Reserved

These bytes are reserved for future expansion.

Byte No. 74. Directory track number
Byte No. 75. Directory sector number

These two bytes point to the location in the directory where the current file is listed. The directory begins on track 0 sector 5, but may extend to other tracks if track 0 is filled.

Bytes No. 76-77. Reserved

These bytes are reserved for future expansion.

Bytes Nos. 78-79. Current or desired sector number

This byte indicates the position of the current sector within the file. In sequential files, the first sector of a file is sector number 1, and so on. In random files the first two sectors, which contain the file map, are number 0, and sector 1 is the first data sector of the file. (See Chapter 13 for further information on random files.)

Bytes No. 80-95. Reserved

These bytes are reserved for future expansion.

Byte No. 96. Beginning of data area

The 256 bytes starting at byte 96 contain the data for an entire disk sector. (Use of a 608-byte FCB leaves another 256 bytes at the end of this data area, thereby allowing for future expansion to 512-byte sectors.)

10. SK*DOS FUNCTIONS

SK*DOS has a large number of subroutines and functions which can be called from user programs. Some of these are actual part of the File Control System; others are simply routines which the FCS itself uses and which are useful to the typical programmer.

This chapter documents these routines and shows how they are used. All of these routines are accessed through the 'exception vectors' of the 68xxx processor.

68xxx processors have a number of 'traps' which trap undefined or illegal operations, and cause a return to a supervisor or operating system via a set of exception vectors in low memory. A full description of this system is beyond the scope of this manual, and we suggest you get the Motorola literature for your processor, or one of the many textbooks on 68xxx programming, for more information.

One of the undefined or illegal 68K operations which causes a trap is any machine language instruction beginning with $A; Motorola literature refers to these as "Line 1010" instructions. Whenever any such instruction is encountered, the 68K CPU does a trap, via one of its exception vectors, to SK*DOS. Thus SK*DOS uses these instructions within user programs to call functions within SK*DOS.

Within SK*DOS, the second byte of each such instruction is used to select a particular function to be performed. For example, the instruction $A001 is used to read a byte from a file, $A002 writes a byte, and so on. A user program calls such a function with the instruction

```
DC    $A0xx
```

where the xx is simply replaced by the number of the function desired.

To avoid the necessity of remembering the numeric code for each particular function, the SK*DOS disk includes a file called SKEQUATE.TXT which provides a series of EQUates which define the exact numeric code for each function. Hence only the function name given in the following descriptions need be remembered. This file may be included as a library file in any user programs you write with the instruction

```
LIB  SKEQUATE   SKEQUATE file included as library
```

Once so defined, the names in the SKEQUATE file can be used in the DC line, as in

```
DC    FREAD
```

Each of the functions listed in this chapter **always** preserves registers D0 through D4, and A0 through A4, and generally **never preserves** registers D5 through D7 and A5 through A6. Arguments passed to SK*DOS are generally passed **in** D4 or A4, as applicable, and arguments passed back to the user program are generally in D5 or A5, as **appropriate.** In addition, all of the following functions always return with A6 pointing to the SK*DOS user- accessible **variable** area (see Chapter 11 and VPOINT for a fuller explanation.)

The functions listed in this chapter are divided into two groups:

A. Functions which do not involve reading or writing to the disk

B. Functions which do involve writing or reading disks

GROUP A. NON-DISK FUNCTIONS

The following SK*DOS functions do not directly involve disk operations:

COLDST Cold start

COLDST is the only function which is not called through an $Axxx trap. Instead, it is the entry point that is used when SK*DOS is first loaded from disk and executed. Entering at COLDST erases all pointers and completely initializes SK*DOS to the beginning. User programs should not use this entry point, especially when files are open, as entering at COLDST causes SK*DOS to 'forget' all its open files. This can corrupt the contents of a disk or its directory. Nevertheless, COLDST may be useful in special applications. Keep in mind, however, that the precise address of COLDST depends on the particular system SK*DOS is run on. You may determine the appropriate address for your system by using the LOCATE command (with the - option) to determine the load address of SK-DOS.SYS. COLDST should be entered with a JMP instruction.

WARMST $A01E Warm start

WARMST is the re-entry point to be used by user programs. Re-entering at WARMST closes all open files and thus helps to insure the integrity of the directory. SK*DOS then prints its prompt and looks for a command to be processed by the Command Processor System.

RENTER $A025 Re-enter SK*DOS

This routine re-enters the SK*DOS command processor system at the point where it processes a command line. It is used when it is desired to continue processing the remainder of a command line (such as after the O or P commands.)

VPOINT $A000 Point to SK*DOS variable area

This routine returns the address of the SK*DOS variable area in A6. Indexed addressing via A6 may then be used to access those variables in SK*DOS of interest to user programs. VPOINT may not be needed in most programs, since all SK*DOS function calls also return this address in A6. The variables which may be accessed are listed in Chapter 11.

GETCH $A029 Get input character with echo

GETCH is used to get an input character from the keyboard; it returns with the character in D5. All valid keyboard character codes can be input, but the parity bit (bit 7) is changed to a 0 for all input. The character is echoed to the output.

INNOEC $A02A Get input character without echo
INNOE1 $A043 Get input character without echo (bypass typeahead)

INNOEC is just like GETCH, but characters are not echoed to the output, and the parity bit is not cleared. Thus INNOEC can be used for 8-bit input, whereas GETCH only reads 7 bits. INNOE1 is similar, but bypasses typeahead (if implemented on the system - see Chapter 14.)

PUTCH $A033 Output character

PUTCH is used to output a character from D4 to the output terminal.

INLINE $A02C Input into line buffer

SK*DOS maintains a 128-byte line buffer which it uses to parse commands to its own Command Processor System. The INLINE routine is used to enter an entire line of text from the keyboard into this line buffer, and may also be used by user programs. A line is normally ended with a CR character ($0D), which is placed at the end of the entered text. Hence the maximum text line which can be entered into the 128-byte buffer is 127 bytes long. This routine permits erasing errors with the backspace key. The Control-X key erases an entire line and starts over. The Control-A key re-displays the entire previous line in the line buffer, from the current cursor position to the previous end of line (carriage return) and can be used to repeat all or any part of a previous line.

PSTRNG $A035 Print CR/LF and string

PSTRNG is used to output an entire string of text to the terminal. The string is preceded by a carriage return and line feed, meaning that the text begins on a new line on the screen. On entry, A4 should point to the first character to be printed, and the string should end with an 04 byte to denote end of data.

PNSTRN $A036 Print string (WITHOUT CR/LF)

PNSTRN is used to output an entire string of text to the terminal. Unlike PSTRNG, however, it is not preceded by a carriage return and line feed. On entry, A4 should point to the first character to be printed, and the string should end with an 04 byte to denote end of data.

CLASFY $A02E Classify alphanumeric characters

This routine is used to classify characters in D4. If the character is a letter or digit, then the C (carry) bit of the condition code register is cleared; otherwise, it is set.

PCRLF $A034 Print CR/LF

This routine prints a carriage return / line feed; that is, it forces the cursor to the next line so that subsequent input or output will occur at the beginning of a new line.

GETNXT $A02D Get next character from buffer

This routine is used to get the next character from the 128-byte input buffer used by SK*DOS. This character is returned in D5 and also placed in the CURRCH location in SK*DOS; the previous character which was in CURRCH is placed into PREVCH so that user programs have access to the last two characters taken from the line buffer. This routine automatically calls CLASFY, so that the carry bit can be used to indicate whether the current character is alphanumeric or not. If the line buffer contains a string of spaces, then GETNXT will return only one space. GETNXT will continue fetching characters until it gets to the end of the line, at which time it will continue to output the end of line ($0D (CR) or ENDLIN) if it is called again, and will also set the carry bit to indicate a non-alphanumeric character. This routine uses the LPOINT pointer to keep track of the next character to be taken from the buffer. This pointer is normally set to the beginning of the buffer after a line is input from the keyboard with INLINE, and is incremented by one each time a character is fetched from the buffer, so that it always points to the next character to be fetched. At the end of a line, it always points to the CR character.

RESIO $A020 Reset I/O pointers

RESIO resets I/O vectors to their initial states. For example, if output is vectored to a disk file, a call to RESIO returns output to the terminal. In general, RESIO resets console I/O vectors to their normal conditions. RESIO is called during WARMST so that SK*DOS always returns to a known state upon return from a user program.

RESTRP $A021 Reset trap vectors

RESTRP resets the 68K processor's trap vectors to those initially used by SK*DOS at boot. (see TRPFLG for further information.) RESTRP is called during WARMST so that SK*DOS always returns to a known state upon return from a user program.

GETNAM $A023 Get file name into FCB

This routine is used to take a file specification from the input buffer and place it into the appropriate bytes of an FCB. At entry, A4 should point to the FCB to be filled, and the LPOINT line buffer pointer should point to the beginning of the file specification in the line buffer. As the file specification is parsed, the drive number will be placed into the drive number location of the FCB (unless no drive number is specified, in which case the working drive will be used). The directory name, file name and the extension, if present, will also be placed into the FCB; if the directory name is missing it will be replaced by the default directory, and a missing name or extension will be replaced by zeroes. If the file specification has no errors, then the carry will be cleared; else it will be set. The file specification in the line buffer may end with a space, comma, CR, or ENDLIN character; if a space or comma, then LPOINT will point to the next character after it, if a CR or ENDLIN, then LPOINT will point to the CR or ENDLIN character. Errors will place error code 21 (illegal file name) into the FCB and set the carry.

LOADML $A022 Load open machine language file

This routine is used to load a machine language file into memory at its normal load address (which is equal to the address listed in the file, plus the OFFSET address, except that the OFFSET address is not added if LASTRM contains a minus sign.) LOADML is normally used by the memory resident GET command to fetch programs prior to execution. Prior to entering LOADML, the user program should use the USRFCB (user FCB) to open the file to be loaded. The file is then loaded, and its transfer address is stored in the EXECAD location. If there is no transfer then XFERFL is set to 0; else it is non-zero. Any transfer address found is stored in the EXECAD location. Errors such as error 4 (file not found) cause an immediate return to the calling program with a non-zero condition and the FCB indicating the error; read errors once a file is found immediately abort the program, close all files, and return to SK*DOS warm start.

DEFEXT $A024 Default extension

This routine is used to enter a default extension into an FCB if the file specification already in the FCB does not contain one. Before entering, the user program should point A4 to the beginning of the FCB, and should place into D4 a numeric code which indicates which default is desired. The codes are as follows:

0 = BIN	3 = BAS	6 = SCR	9 = DIR	12 = BAT
1 = TXT	4 = SYS	7 = DAT	10 = PRT	13 = SRC
2 = COM	5 = BAK	8 = BAC	11 = OUT	14 = PIP

OUT5D $A038 Output 5 decimal digits

This routine outputs a decimal number of up to five digits. Before entering, the calling program should place into D4 the unsigned binary word to be printed, and set D5 to zero if the number is to be printed without leading spaces or zeroes, or to nonzero if the number is to be printed with leading spaces.

OUT10D $A039 Output 10 decimal digits

This routine outputs a decimal number of up to ten digits. Before entering, the calling program should place into D4 the unsigned binary long-word to be printed, and set D5 to zero if the number is to be printed without leading spaces or zeroes, or to nonzero if the number is to be printed with leading spaces.

OUT2H $A03A Output 2 hex digits

This routine prints the two-digit hexadecimal number that is in the right-most byte of D4 on entry.

OUT4H $A03B Output 4 hex digits

This routine prints the four-digit hexadecimal number that is in the right-most word of D4 on entry.

OUT8H $A03C Output 8 hex digits

This routine prints the eight-digit hexadecimal number that is in D4 on entry.

PERROR $A037 Print error code

When an error is encountered by the FCS while using an FCB, user programs should do a call to this routine to print the error code. PERROR should be entered with A4 pointing to the beginning of the FCB, and the error code in byte 1 of the FCB. The error codes are listed in Appendix E. The error code is printed as a number; in addition, if the system disk contains the file ERRCODES.SYS, SK*DOS will read a one-line text description from this file and print it alongside the numeric code to explain the error's meaning.

TOUPPR $A031 Convert lower case to upper (in D5!)

Converts a lower case character **in D5** into upper case. Primarily for use right after GETCH or GETNXT, if only upper case letters are desired.

HEXIN $A02F Input hexadecimal number

This routine inputs a hexadecimal number from the line buffer and places it in D5. Before entering, the calling program should make sure that LPOINT points to the first digit of the number to be input; at the end, LPOINT will be left pointing as described earlier for GETNAM. On output, D6 is non-zero if a number was actually found, and the carry bit is set if an invalid character was found in the number. (It is possible for both D6 and the carry to be zero if HEXIN encounters a delimiter such as a space, comma, CR, or ENDLIN immediately on entry.) If a number is not found the number returned is zero; if the number is greater than $FFFFFFFF, then only the last eight hex digits are returned.

DECIN $A030 Input decimal number

This routine is similar to HEXIN, but inputs a decimal number rather than a hexadecimal one.

EXECSD $A01F Execute a SK*DOS command

This entry point allows a user-written program to call SK*DOS as a subroutine and have it execute a command line placed into the line buffer. On entry, A4 should point to the beginning of the command (usually at the beginning of the line buffer), and the command should end with a CR or ENDLIN character. If the command line in turn executes a disk-resident program, then that program should end with a DC WARMST instruction to return to SK*DOS. SK*DOS, in turn, knowing that the program was called from another user program, will return control to the user program. The user should be careful not to call a program which will overlay part of the calling program in memory.

STATUS $A02B Check keyboard for character
STATU1 $A042 Check Keyboard for character (bypass typeahead)

This routine allows a user program to check whether a character is being entered from the keyboard. If no character is being entered, the Z bit in the condition code register is set; if a character is being entered, then the Z bit is clear. All other registers are preserved. STATU1 is similar, but bypasses typeahead (if implemented - see Chapter 14.)

INTDIS $A040 Disable Interrupts

This routine masks interrupts (to level 7), thereby preventing the CPU from being interrupted by level 0 through 6 interrupts. This SK*DOS call is intended only for use by advanced programmers, and then only in systems programs such as FORMAT.

INTENA $A041 Re-Enable Interrupts

This routine restores interrupts to the same status as existed before the last previous INTDIS call. Make sure not to use INTENA unless it has been preceded by a INTDIS. This SK*DOS call is intended only for use by advanced programmers, and then only in systems programs such as FORMAT.

FINDEV $A012 Find device from name

This function converts a device name (such as CONS for console) to a device number (plus $10). For example, to find out whether PRTR has been installed as a printer driver, place the name PRTR (followed by seven $00 bytes to erase the remaining name and extension bytes) into the file-name bytes of an FCB, and call FINDEV. If PRTR is not installed, then SK*DOS will return error 4 (not found); if it is installed, then SK*DOS will place the device number (plus $10) into the logical drive-number byte. For example, if PRTR is device 2, then FINDEV will return $12 in byte 3 of the FCB.

GETDNT $A03F Get date and time

If the system contains a clock/calendar IC, then this function returns the current date and time in D5 and D6 as shown below:

```
D5: WWMMDDYY
      |  |  |  +- year in hexadecimal
      |  |  +----- day in hex
      |  +-------- month in hex
      +----------- day of week (00=none, 01=Sunday, 02=Monday, ...)
D6: 00HHMMSS
      |  |  |  +- seconds in hex
      |  |  +----- minutes in hex
      |  +-------- hours in hex (24-hour time)
      +----------- always zero
```

If no clock/calendar IC is available, then the day of week byte of D5, and all of D6, are zero, and only the date (month/day/year) is returned (obtained from the date typed in by the user upon booting).

ICNTRL $A028 Input Control

OCNTRL $A032 Output Control

These two functions permit device driver selection, special characters to be passed to or from a device driver, and other device functions. See Chapter 14 for a more complete explanation.

FLUSHT $A044 Flush Type-ahead buffer, if any.

This function flushes (empties) the keyboard typeahead buffer (if implemented - see Chapter 14.)

FNPRNT $A045 Print file-name

This function formats and prints the directory and file-name (but not drive number) pointed to by A4. In memory, the name should consist of 11 bytes, 8 for the directory and name, and 3 for the extension, with no period between them. In addition, D4.B is used to specify whether to provide spaces for missing items. If D4.B is zero, then the name might be printed as just NAME.EXT; if D4.B is not zero, then there would be two spaces before NAME (leaving space for a possible directory name), and four spaces between NAME and the period (leaving space for an 8-character name). Furthermore, if the extension is missing (zero in memory) then using a non-zero D4.B would leave three spaces after the period.

GROUP B. DISK FUNCTIONS

The following SK*DOS functions involve reading or writing disks.

FCSINI $A01B Initialize File Control System

This function should not normally be used by user programs as it can result in corruption of the disk. It totally initializes the system - disk drivers are initialized, all open files are forgotten and left open, etc.

FCSCLS $A009 Close all open files

This routine allows user-written programs to close all open files without actually knowing which they are. If FCSCLS detects an error, then it prints error 13 (error in closing file), clears the Z bit in the condition code register, and returns with A5 pointing to the FCB which was being closed when the error was detected. When FCSCLS detects an error, it does not close the remaining files; hence its routine use to close files is not encouraged. Instead, users should close each file separately.

NOTE: Each of the following Group B. functions requires that the calling program must point A4 to the FCB to be operated on, and then do a call to the appropriate function specified below. If the operation is finished without error, then FCS returns with the Z bit set. If an error is detected, then the Z bit is cleared and the error code is in byte 1 of the FCB. If no error is detected, this byte is not necessarily 0, and should not be tested unless the Z bit indicates an error.

The normal call to these functions is thus

```
LEA     _____,A4        Point to the File Control Block
DC      <function>      Call FCS to perform operation
BNE     ERROR           Go process error if detected
```

Note that it is required that A4 point to the beginning of the FCB when FCS is called. Since the contents of A0 through A4, and D0 through D4, are preserved upon return from the FCS, so A4 will still be pointing to the FCB upon return.

If the FCS is called with an unimplemented operation code, the FCS will print out an error message and return to SK*DOS.

The following descriptions include typical error codes that may be generated on specific operations. In addition, most of the operations may also result in disk read or write errors due to hardware problems.

In all cases, if no error occurs, then the FCS returns with the Z bit of the condition code register set. If an error does occur, then the Z bit is cleared and byte 1 of the FCB (as well as location ERRTYP) contains the code for the error that occurred. Error codes are listed in Appendix E.

FREAD $A001 Read the next byte from file

FWRITE $A002 Write the next byte to the file

Most FCS read or write operations are sequential; these functions are used to read or write the next sequential byte or character in the file. During a read, the next byte from the file is read from the sector currently in the FCB (using the data pointer in byte 49) and returned in D5, while during a write the character in D4 is written to the file. Since the file is read or written on the disk an entire sector at a time, this function actually buffers the data through the sector buffer (bytes 96- of the FCB). Hence no actual disk read or write will generally occur for most FREAD or FWRITE 0 calls. When an actual disk read or write is required, SK*DOS will handle that automatically without user intervention.

FOPENR $A005 Open a file for read

This function opens a file for reading. Before calling the FCS, the calling program must insert the drive number, name, and extension of the desired file into bytes 3 through 14 of the FCB. The FCS will take care of initializing all other parts of the FCB. If the requested file is not on the disk, the FCS will return error code 4 (file does not exist). When the FCS is finished opening the file, it prepares the file for sequential reading next, and assumes that space compression will be used.

FOPENW $A006 Open a file for write

This function opens a file for writing. Before calling the FCS, the calling program must insert the drive number, name, and extension of the desired file into bytes 3 through 14 of the FCB. The FCS will take care of initializing all other parts of the FCB. If the specified disk already has a file with the specified name, the FCS will return error code 3 (file already exists). This function opens a sequential file, but it may be changed to a random file by storing a non-zero number into byte 26 of the FCB after opening the file, but before writing any data into it. (See Chapter 13 for further information on random files.)

FOPENU $A007 Open a file for update

This function opens a random file for reading or updating. Once the file is open, you may do one of the following:
a. Use FRRECD to position to a particular sector of the file.
b. Use FRBACK to backup to the preceding sector.
c. Use FRGET to read a particular byte from the currently selected sector.
d. Use FRPUT to write a particular byte to the currently selected sector.
e. Use FREAD to sequentially read the sector, starting with the first. You may read as many bytes as there are in the file. (If executed after opening the file for update, FREAD will start reading at the beginning of the file.)
f. Use FRRECD to extend the file.
g. Use FCLOSE to close the file.
h. The only way to write past the end of a sector into the next sector is to use FRRECD to position to the next sector.

(See Chapter 13 for further information on random files.)

FCLOSE $A008 Close file

This function closes a file currently opened for reading, writing, or updating. No operation is performed on read files, or on write files which were never written to, other than removing them from the chain of file pointers. When a write file is closed, any data remaining in the data area of the FCB is written out to the disk, and both the directory and the file sector map are updated to indicate the correct track and sector numbers of the last sector and the file size. The system information sector (SIS) on track 0 sector 3 is also updated. When a random file open for updating is closed, the current sector is written out to disk if data has actually been written to it.

FREWIN $A00A Rewind file

This function can only be performed on a file which is open for reading, and will result in Error 1 (FCS function code error) if attempted on a file open for writing. The Rewind function is used to start reading a file from the very beginning, and is equivalent to closing the file and then immediately opening it for reading again.

DIROPN $A00B Open directory file

This function prepares the current FCB to read directory entries with DSREAD. Before using this function, the calling program should place the drive number into byte 3 of the FCB; no other initialization is required. This function does not actually do any reading, but merely prepares the FCB for a subsequent read with DSREAD. It is primarily used by the FCS itself, and will not usually be used by user programs.

SISOPN $A00C Open system information sector

This function prepares the current FCB to read the SIS with DSREAD. Before using this function, the calling program should place the drive number into byte 3 of the FCB; no other initialization is required. This function does not actually do any reading, but merely prepares the FCB for a subsequent read with DSREAD. It is primarily used by the FCS itself, and will not usually be used by user programs.

DSREAD $A00D Read directory or system information sector

This function must be preceded by either DIROPN (open directory) or SISOPN (open system information sector) and reads the first (or next) entry in the directory or SIS, respectively, into the first 96 bytes of the FCB (it also reads the entire sector into the sector buffer area of the FCB). When used on the SIS, only one read should be performed since only one entry exists in the SIS; when used on the directory, up to ten reads can be performed on any one sector since there are ten directory entries per sector. Upon the eleventh read, DSREAD will automatically read the next sector of the directory. Error 8 (input past end of file) will be returned at the end of the directory. Note that all entries are read, even deleted entries (indicated by a $FF as the first character of the file name) or unused entries (indicated by a 00 as the first character.)

DSWRIT $A00E Write directory or SIS entry

This function writes a directory or SIS entry from the first 96 bytes of an FCB back into its appropriate position in the sector buffer, and then writes the sector buffer back to the disk. Because of the need to properly set up a number of pointers, this function should only be used after DSREAD, which in turn should only be used after DIROPN or SISOPN.

FDELET $A00F Delete a file

This function deletes a file name from the directory and returns its used sectors to the chain of free sectors maintained in the System Information Sector (track 0 sector 3). Before using this function, the calling program must place the drive number, file name, and file extension of the desired file into bytes 3 through 14 of the FCB. It returns error 4 if the file name is not found, along with the Z bit of the condition code register cleared.

FRENAM $A010 Rename a file

This function renames an existing file; before calling the FCS, the user's program must place the old file specification into bytes 3 through 14 of the FCB, and the new name and extension into bytes 52-62 of the FCB (temporary name buffer 2). If there is an error, the Z bit is cleared and the FCS returns one of the following codes in byte 1 of the FCB: error 4 (old file does not exist), or error 3 (new file name already exists).

FSKIP $A011 Skip current sector

This function skips the current sector and goes to the next sector of the current file. When a file is being read, the FCS simply skips the remaining data in the current sector and prepares to read the next sector. On writing, the FCS fills the remainder of the current sector with zeroes, writes it out to the disk, and prepares to write into the next sector; if, however, the FCS is already pointing to a new sector (but has not yet written into it) then this function is ignored.

FRRECD $A014 Select a specified random sector

This function allows you to select a specified random sector of a random file. This function can only be used after an existing random file is opened for update with FOPENU. To use this function code, place the two-byte sector number of the desired sector into bytes 78 and 79 of the FCB and and then call the FCS. If the desired sector number is 0, then the first sector of the file map will be read; if the desired sector number is larger than the current size of the file, the file will be extended so that the desired sector is the last sector in the file, and all new sectors will be filled with zeroes. (See Chapter 13 for further information on random files.)

FRBACK $A015 Backup to previous sector

This function allows you to backup to the previous random sector of a random file. Read the description of FRRECD, as this function is similar. You cannot backup from sector 1 (the first data sector of the file) back to sector 0 (the first file map sector), and any attempt to do so will generate error 24 (invalid sector number). (See Chapter 13 for further information on random files.)

FRGET $A016 Read a random byte

This function allows you to read a specified random byte from a random file currently opened for update. To select the byte, place its number, a value from 4 to 256, into byte 51 of the FCB. (Note that there are only 252 data bytes per sector, and they are numbered from 4 through 256.) The byte will then be read from the currently selected sector. (See Chapter 13 for further information on random files.)

FRPUT $A017 Write a random byte

This function allows you to write a specified random byte to a random file currently opened for updating. To select the byte, place its number, a value from 4 to 256, into byte 51 of the FCB. (Note that there are only 252 data bytes per sector, and they are numbered from 4 through 256.) The byte will then be written to the currently selected sector. (See Chapter 13 for further information on random files.)

SREAD $A01C Read a single sector

This function provides direct access to the disk read routine. The user program must provide the drive number in byte 3 of the FCB, and the track and sector numbers in bytes 34-35. Upon exit, the data from the desired sector begins at byte 96 of the FCB. If an error is encountered, the Z bit of the condition code register is cleared and byte 1 of the FCB contains one of the following error codes: error 9 (disk read error), error 14 (disk seek error), or error 16 (drive not ready).

SWRITE $A01D Write a single sector

This operation provides direct access to the disk write routine. The user program must provide the drive number in byte 3 of the FCB, the track and sector numbers in bytes 34-35, and the data to be written beginning at byte 96 of the FCB. If an error is encountered, the Z bit of the condition code register is cleared and byte 1 of the FCB contains one of the following error codes: error 10 (disk write error), error 11 (write protected disk), error 14 (disk seek error), error 16 (drive not ready), or error 29 (disk verify error).

FDRIVE $A01A Find next drive number

This function is used to find the next available drive number. On entry, you must place either the number $FF or a valid drive number into byte 3 (the drive number byte) of the FCB. The FCS will then return with the next available drive number. The FCS will start with the next higher drive number; since $FF is equivalent to -1, entering with this value will start with drive 0. Searching will continue up until the current value of MAXDRV; if no ready drive is found, the FCS will return error 16 (drive not ready) in byte 1 of the FCB and also set the carry bit; otherwise the carry bit is cleared.

DIREST $A026 Disk Restore

This function does a restore on the drive pointed to by the drive number in the current FCB; that is, the head on the current drive is retracted to track 0. This function is only implemented for the floppy disk, and is used specifically by the FORMAT routine; it should not be used by any other programs.

DISEEK $A027 Disk Seek

This function causes the drive indicated in the current FCB to seek (i.e., move the head to) the 'current track number' given in the FCB. No checking is actually done, other than checking that the drive number is valid. This function is only implemented for the floppy disk, and is used specifically by the FORMAT routine; it should not be used by any other programs.

11. USER-ACCESSIBLE VARIABLES

Many of the SK*DOS variables are of use to a programmer writing programs to run under SK*DOS. These variables are of two types:

1. User variables which are often needed by application programs running under SK*DOS. These are described in this Chapter.

2. System variables which are generally needed only by systems programmers implementing SK*DOS on a new system, or modifying major operating parameters. A few of these are described in Chapter 13, though most such system variables are described in the optional Configuration Manual, not in this Users' Manual.

The precise location of User variables may change between various versions of SK*DOS. Any call to an SK*DOS function, however, returns in A6 a pointer to the beginning of this variable area. In particular, the call to VPOINT (see Chapter 10) specifically exists to return the address of this variable area in A6. Each of the locations described below can be referenced using indexed addressing with reference to A6. This may be done either by referring to the numeric offset given in the descriptions below, or by using the SKEQUATE file on your SK*DOS disk as a library file in your assembly language programs as follows:

```
LIB SKEQUATE  SKEQUATE file included as library
```

Once this is done, you may refer to variables by their symbolic name. For example, the following two lines are equivalent:

```
LEA 0(A6),A4       Uses absolute offset
LEA USRFCB(A6),A4  Uses offset defined in SKEQUATE
```

NOTE: In the following, the abbreviation DN stands for Device Number, and is used for those variables whose addresses are different for different devices. For example, PLINES is listed as 3322+80*DN(A6). For device number 0, the address is 3322(A6); for device 1 it is 3322+80*1(A6), or 3402(A6), and so on.

USRFCB 0(A6) User FCB (608 bytes)

This area is an FCB which is used by SK*DOS for its own internal operations. This is done in such a way, however, that other programs which require an FCB can also use this 'user' FCB without interfering with SK*DOS. This can save the effort of having to declare memory for a separate FCB. (As mentioned earlier, the USRFCB is 608 bytes long although at this time only 352 bytes are actually used. The extra 256 bytes are left for future expansion.)

LINBUF 608(A6) Line buffer (128 bytes)

The line buffer is a 128-byte buffer which is used by SK*DOS for holding and processing commands. It is, however, also accessible to user programs through the INLINE routine (which enters text from the keyboard into the buffer) and the GETNXT, GETNAM, and other routines (which take data from the buffer). In particular, note that whenever a user program is called from the keyboard while in SK*DOS, any remaining text entered after the program name in the command line is still in the line buffer, and may be recovered by the user program. For example, when the LIST program is called from the keyboard with a

SK*DOS: LIST TEXT

command, when the LIST program begins, the line buffer pointer LPOINT points to the first letter of the word TEXT. The LIST program can access the name with several subroutines, such as GETNAM. This is a convenient way of getting and passing arguments to programs directly from the command line.

BACKSC 736(A6) Backspace character ($08)

DELETC 737(A6) Delete character ($18)

ENDLNC 738(A6) End of line character ($3A)

ESCAPC 746(A6) Escape char ($1B)

REPEAC 749(A6) Repeat character ($01)

These locations contain the backspace character ($08 or control H), delete character ($18 or control X), end of line character ($3A or colon), escape ($1B), and repeat ($01 or control-A), respectively. These locations are used by SK*DOS in console input routines, and can be changed by user programs or the DOSPARAM command. The DELETC character may be used to delete an entire line and return to the beginning, while the ENDLNC character, normally a colon, is accepted much the same as a carriage return ($0D) as an end-of-command delimiter. The ENDLNC character, however, can also be used to separate multiple commands on one line. The ESCAPC character is used to halt output to the terminal. Output may be restarted with another ESCAPC, or else may be aborted with a carriage return (not ENDLNC). The REPEAC character, normally $01 or Control-A, is used to repeat the last-used console command.

PLINES 3322+80*DN(A6) Number of printed lines per page

SLINES 3325+80*DN(A6) Number of skipped lines between pages

These two locations are initialized by SK*DOS at 0, which disables them. If PLINES is non-zero, then output to the terminal will stop after PLINES continuous lines of output, skip SLINES blank lines, and then continue. A typical application is to make PLINES equal to a decimal 56, and make SLINES equal to a decimal 10, for paged output to a printer. Printed output would then have 56 continuous lines of print, and 10 skipped lines which step over the perforation between pages. These constants may be changed with the DOSPARAM command.

PAUSEB 3326+80*DN(A6) Output pause control byte

If PAUSEB is non-zero, and PLINES is non-zero, then terminal output will pause after each PLINES lines, and resume when the escape key is pressed on the keyboard. PAUSEB is normally initialized at $FF, thereby enabling the pause, but since PLINES defaults to 0, no pause actually takes place.

PWIDTH 3323+80*DN(A6) Page column width

This byte is used to control the page width of output to the terminal or printer. When output goes to the right of the PWIDTH column on the terminal or printer, SK*DOS will issue a carriage return / line feed at the next space character. PWIDTH may be set with the DOSPARAM command. SK*DOS normally initializes PWIDTH to 0, which disables it. To use PWIDTH, you should set it to a value approximately 10 less than the actual screen or paper width, so as to leave room for any long words at the end of the current line. (PWIDTH works in conjunction with the OCOLUM and SPECIO variables described below.)

SPECIO 792(A6) Special I/O Indicator

When SPECIO is nonzero, the PWIDTH value is ignored by SK*DOS. SPECIO is initialized to zero at warm-start and by RESIO.

OCOLUM 3328+80*DN(A6) Current output column

This byte indicates the current output column on the terminal or printer. It is used with the PWIDTH variable, and reset to zero at the beginning of every line, or by the printing of any control character.

NULLWT 3324+80*DN(A6) Null wait constant

Some terminals or printers make errors if a carriage return or line feed character is immediately followed by printable characters. In that case, NULLWT may be used to insert a short delay. It is normally initialized to 00 or no delay, but may be changed by the user (via the DOSPARAM command). The wait delay depends on the CPU clock speed, but is approximately equal to 0.01 second times the value of NULLWT.

SYSDIR 744(A6) System default directory

WRKDIR 745(A6) Working default directory

Both of these locations are initialized at 0. They are used as default directories. SYSDIR is used as the default for loading any disk-resident command, while WRKDIR is used as a general default by the GETNAM routine whenever a directory is not specified as part of a file spec. A value of 0 refers to the main or *root* directory, while values of $41 through $5A refer to subdirectories A/ through Z/ (since these are the ASCII equivalents for the letters A - Z.)

SYSTDR 747(A6) System default drive

WORKDR 748(A6) Working default drive

Both of these locations are initialized at 0. They are used as default drive numbers. SYSTDR is used as the default for loading any disk-resident command, while WORKDR is used as a general default by the GETNAM routine whenever a drive number is not specified as part of a file spec.

CMONTH 750(A6) Current date - month

CDAY 751(A6) Current date - day

CYEAR 752(A6) Current date - year

The above three locations hold the month, day, and year entered from the keyboard when SK*DOS is first cold started. This date is used by SK*DOS when writing files on the disk, and may also be accessed or changed by user programs. All three bytes are binary numbers, and CYEAR contains only the last two digits of the year; for example, in 1990 the registers contain 90, or a hexadecimal $5A.

LASTRM 753(A6) Last terminator

This byte contains the last terminator encountered by the GETNXT routine from the line input buffer.

COMTAB 754-757(A6) Pointer to command table (long word)

SK*DOS contains several memory-resident commands (such as XEQ and GET); users may add additional memory-resident commands, and let SK*DOS know about them via COMTAB by putting into COMTAB a pointer to a table which lists the added commands. This table is searched by SK*DOS after its own command table, but before it looks for disk resident commands. Each entry in the user command table should consist of (a) the command name of up to eight letters (with no extension), (b) a zero byte to signal the end of the name, and (c) a four-byte address pointing to the command program. An extra zero byte at the very end signals the end of the table. The command should end with either an RTS or DC WARMST instruction.

LPOINT 758-761(A6) Pointer to line buffer (long word)

LPOINT points to the next character to be obtained from the line buffer. When the buffer is first filled (with INLINE), LPOINT points to the first character in the buffer. Each time another character is obtained from the buffer, LPOINT is incremented so that it points to the next byte. When the pointer gets to the CR code at the end of the line, it then remains pointing to the CR. When an entire name or number is fetched from the buffer, such as by GETNAM, then at the end of the routine LPOINT points to the first character past the delimiter (such as a space or comma), or to the delimiter itself (if CR).

BREAKA 762-765(A6) Break (Escape) address (long word)

As indicated earlier, terminal output can be interrupted by pressing the ESCAPE key, at which time SK*DOS waits for a second character. If the second character is again an ESCAPE, then output resumes; if the second character is a carriage return ($0D) character, then SK*DOS will abort the program. This return is handled through the return address in BREAKA, which is initialized by SK*DOS to point to WARMST. User programs may also use BREAKA to return to SK*DOS in that way. More commonly, user programs may store a different address in BREAKA to force a return elsewhere when the return key is pressed.

CURRCH 766(A6) Last character read from buffer

PREVCH 767(A6) Previous character read

As characters are fetched from the line buffer by SK*DOS routines (such as GETNXT), these two locations hold the latest character fetched (CURRCH) and the previous character fetched (PREVCH).

EXECFL 774(A6) Execution address flag

This location is non-zero when location EXECAD contains a valid execution address for a machine language program, and is zero when such a valid execution address does not exist. If a command such as XEQ is executed when there is no valid address, then SK*DOS will print error 28 (missing transfer address.)

EXECAD 776-779(A6) ML execution address (long word)

These four bytes hold the transfer address obtained when a machine language file is loaded from the disk (including the value of OFFSET, if used). This location is also used by the XEQ command to execute the last-loaded program.

ERRTYP 782(A6) Error type

This byte contains the number of the last error detected by the File Control System.

FOADDR 784-787(A6) File output address vector

FIADDR 788-791(A6) File input address vector

These two addresses are used for redirection of standard output or input, respectively. If they are zero, no redirection is done. If redirection is desired, then one (or both) of the above vectors may contain the address of an FCB currently open for writing or reading, respectively.

CMFLAG 793(A6) Command flag

This location indicates whether the Command Processor System is processing a keyboard command (when 0) or a command passed to it from a user program (when non-zero).

MEMEND 796-799(A6) Last usable memory address (long word)

When SK*DOS is initially booted, it does a memory test to determine how much memory is installed in the system, and then stores the address of the last memory location in MEMEND. OFFSET and MEMEND together therefore define the lower and upper limits, respectively, of free user memory. User programs can check these locations to determine how much user memory is available, or can change the contents to set aside memory for themselves.

ECHOFL 800(A6) Input echo flag

This location tells the character input routine whether to echo output to the output port. A non-zero value (initialized to $FF at warm-start) enables echo.

FNCASE 801(A6) File Name case flag

This location determines whether lower-case file names are allowed either as disk-resident command names, or as file names processed by the GETNAM function. The default value is $DF, which allows only upper-case names (and lower-case names are automatically converted to upper case). Lower-case names will be allowed if FNCASE is changed to $FF.

MAXDRV 802(A6) Maximum drive number

This location is used to define the maximum drive number on the system. It is initialized to 03, and the maximum number it may have is 09. SK*DOS will return a drive number error whenever a drive number is specified in a file-spec which exceeds the value of MAXDRV. If your system has more than four drives, then MAXDRV should be increased correspondingly when the appropriate disk drivers are installed.

SEQNO 806(A6) Sequence Number

This byte holds the sequence number assigned to each file written. If the system does not contain a clock-/calendar chip, then this number is written into the disk directory along with the date for the file; if a clock-/calendar chip is interfaced to SK*DOS, then the sequence number is replaced by the time (although it is still calculated and stored in SEQNO.) Since the sequence number is just a single byte, it has a maximum value of 255; then it returns back to 0 and repeats its cycle.

ERRVEC 834(A6) Alternate ERRCODES.SYS vector

ERRVEC allows SK*DOS to get its error messages from a file other than ERRCODES.SYS. If this long-word contains 0, then the normal ERRCODES.SYS file is used; if it is non-zero, then it is assumed to hold a long-word address pointing to an 11-character file specification containing the name and extension of another file to be used. ERRVEC is initialized at 0 when SK*DOS is booted, but is not changed thereafter; hence user programs should be careful to restore it when they are finished using other error files.

DOSORG 838-841(A6) Absolute ORG of SK*DOS

DOSORG contains the actual starting address of SK*DOS in the current system. This information is primarily for systems programmers, as most users will have no need to know absolute memory addresses.

OFFSET 770-773(A6) Offset load address (long word)

OFFINI DOSORG+$18 Initial OFFSET value (long word)

The contents of OFFSET is added to the load address and transfer (or execution) address for all machine language programs loaded from disk by SK*DOS (unless LASTRM contains the ASCII code for a minus sign.) Programs are thus loaded into the address specified in the disk file only if OFFSET is 0 or if LASTRM contains a minus sign. Normally, OFFSET points just above SK*DOS, so that all user programs are loaded into free memory above SK*DOS. Such programs can be executed as long as they are written to be position independent. OFFSET and MEMEND together therefore define the lower and upper limits, respectively, of free user memory.

Since user programs may change OFFSET as they run (to load another program into memory above themselves, for example), SK*DOS resets OFFSET to the value stored in OFFINI each time it does a warm start (it actually sets OFFSET to the next 256-byte boundary above OFFINI to make OFFSET a more convenient number.) OFFSET is thus only a temporary value, valid for the duration of any given program; OFFINI is more permanent.

DEVIN 3274(A6) Current Input device

DEVOUT 3275(A6) Current Output device

DEVERR 3276(A6) Current Error device

These three bytes specify the devices currently being used for input, output, and error displays. They default to device 0 for input and output, and device 1 for the error device, all of which are normally the default CONS console device. User programs can change DEVIN and DEVOUT to go to different devices, but DEVERR should usually remain so that error messages still go to the console.

DEVTAB 3278(A6) Device Descriptor Table

The device descriptor table provides information on currently installed devices; more information is found in Chapter 14.

BAUDRT 3329+80*DN(A6) Baud Rate/100

This byte specifies the baud rate for serial devices, and is not used for others. BAUDRT indicates the baud rate divided by 100; for example, a baud rate of 110 is shown as 01. If no baud rate is specified, the driver may use its own default value.

EOFILC 3330+80*DN(A6) End-Of-File Character

When a device rather than a file is used for input during input redirection, EOFILC defines which character will generate error 8 (end of file). The default is $1A, which is control-Z.

XOFFC 3331+80*DN(A6) X-Off Character

XONC 3332+80*DN(A6) X-On Character

These two bytes specify whether a serial port uses the X-On/X-Off protocol, and which characters it uses. When these characters are 0 (the default), then no X-On/X-Off is used. When they are set to other values (usually $13 or control-S for X-Off, and $11 or control-Q for X-On) then the port will immediately stop all output upon receipt of an X-Off, and will only resume upon receipt of an X-On. (Note that this is different from using ESCape to halt and resume output, since ESC only works for device 0.)

12. PROGRAMMING EXAMPLES

This section shows several examples of how to use SK*DOS in writing programs which access the disk.

THE LIST UTILITY

The LIST program is one of the utilities supplied with SK*DOS. It is called with a command line which includes the name of the file to be listed after the word LIST, as in

```
SK*DOS: LIST TEXT
```

The program reads the file name from the line buffer, opens the file, and reads and prints one character at a time. The following listing shows how this is done.

```
                        * LIST UTILITY FOR SK*DOS / 68K
                        * COPYRIGHT (C) 1986 BY PETER A. STARK
                        *
                        * EQUATES TO SK*DOS
                        *
                        *   THE FOLLOWING WOULD NORMALLY BE IN SKEQUATE.TXT
                00000001 FCBERR  EQU     1               ERROR BYTE
                0000A024 DEFEXT  EQU     $A024           DEFAULT EXTENSION
                0000A008 FCLOSE  EQU     $A008           CLOSE A FILE
                0000A005 FOPENR  EQU     $A005           OPEN FILE FOR READ
                0000A001 FREAD   EQU     $A001           READ NEXT BYTE
                0000A023 GETNAM  EQU     $A023           GET FILE NAME
                0000A034 PCRLF   EQU     $A034           PRINT CR/LF
                0000A037 PERROR  EQU     $A037           PRINT ERROR MSG
                0000A033 PUTCH   EQU     $A033           OUTPUT NEXT CHAR
                0000A01E WARMST  EQU     $A01E           RETURN TO SK*DOS
                        *
000000                          ORG     $0000
                        *
000000 6002     }000004 LIST    BRA.S   START           GO TO START
                        *
000002 0100             VER     DC.W    $0100           VERSION NUMBER
                        *
                        * START OF ACTUAL PROGRAM
000004 A034             START   DC      PCRLF           START ON NEW LINE
000006 4201                     CLR.B   D1              PREV CHAR WAS NONE
000008 204E                     MOVE.L  A6,A0           SAVE POINTER
00000A 284E                     MOVE.L  A6,A4           POINT TO USER FCB
00000C A023                     DC      GETNAM          GET FILE SPEC
00000E 640C     }00001C         BCC.S   NAMEOK          IF FILE NAME OK
000010 197C 0015 0001           MOVE.B  #21,FCBERR(A4)  ELSE IT'S ERR 21
                        *
                        * ERROR ROUTINE
                        *
000016 A037             ERROR   DC      PERROR          PRINT ERROR CODE
000018 6142     }00005C         BSR.S   CLOSE           CLOSE THE FILE
00001A A01E                     DC      WARMST          RETURN TO SK*DOS
                        *
```

```
                        * FILE SPEC WAS OK; DEFAULT TO .TXT
00001C 183C 0001        NAMEOK  MOVE.B  #1,D4           DEFAULT EXT
000020 A024                     DC      DEFEXT          DEFAULT TO .TXT
                        *
                        * NOW ACTUALLY OPEN THE FILE
000022 A005                     DC      FOPENR          OPEN FOR READ
000024 66F0     )000016         BNE.S   ERROR           IF NOT ZERO
                        *
                        * MAIN LOOP TO READ AND PRINT EACH CHARACTER
000026 2848             MAIN    MOVE.L  A0,A4           POINT TO SYS FCB
000028 A001                     DC      FREAD           GO READ NEXT CHAR
00002A 670C     )000038         BEQ.S   CHAROK          GO ON IF NO ERROR
                        *
                        * IF THERE WAS AN ERROR, SEE IF END OF FILE
00002C 0C2C 0008 0001           CMP.B   #8,FCBERR(A4)   END OF FILE?
000032 66E2     )000016         BNE.S   ERROR           NOT END OF FILE
000034 6126     )00005C         BSR.S   CLOSE           CLOSE ON EOF
000036 A01E                     DC      WARMST          RETURN TO SK*DOS
                        *
                        * CONTINUE IF CHARACTER IS OK
000038 0C05 000A        CHAROK  CMP.B   #$0A,D5         IS IT LINE FEED?
00003C 660A     )000048         BNE.S   PRNTIT          NO, PRINT IT
00003E 1C01                     MOVE.B  D1,D6           YES, GET PREV
000040 1205                     MOVE.B  D5,D1           SAVE CURRENT
000042 0C06 000D                CMP.B   #$0D,D6         WAS PREV A CR?
000046 67DE     )000026         BEQ.S   MAIN            YES, SO SWALLOW IT
000048 1205             PRNTIT  MOVE.B  D5,D1           SAVE CHARACTER
00004A 1805                     MOVE.B  D5,D4           READY FOR PRINTING
00004C A033                     DC      PUTCH           AND PRINT IT
00004E 0C04 000D                CMP.B   #$0D,D4         WAS IT RETURN?
000052 66D2     )000026         BNE.S   MAIN            NO, SO CONTINUE
000054 183C 000A                MOVE.B  #$0A,D4
000058 A033                     DC      PUTCH           ADD LINE FEED
00005A 60CA     )000026         BRA.S   MAIN            AND ALSO CONTINUE
                        *
                        * CLOSE SUBROUTINE
                        *
00005C 2848             CLOSE   MOVE.L  A0,A4           POINT TO FCB
00005E A008                     DC      FCLOSE          CLOSE FILE
000060 4E75                     RTS                     RETURN
                        *
                                END     LIST
```

The above example shows a variety of techniques. Note especially how it checks for the end of file. When the read routine detects an error, the error code is fetched from byte 1 of the FCB and examined. If it is an 8 (end of file), then the program simply finishes up. If it is any other error, then it goes to an error routine.

Note also that the program is ORG'ed at $0000. When loaded, however, it will be relocated upward by the current value of OFFSET, so that it resides in memory just above SK*DOS.

THE BUILD UTILITY

The following example shows how to open a file for writing and actually proceed to write into it.

```
                        * BUILD UTILITY FOR SK*DOS / 68K
                        * COPYRIGHT (C) 1986 BY PETER A. STARK
                        *
                        * EQUATES TO SK*DOS
                        *
                00000001 FCBERR  EQU     1              ERROR BYTE
                000002F6 LPOINT  EQU     $2F6           LINE BUFR PTR
                         *   THE FOLLOWING WOULD NORMALLY BE IN SKEQUATE.TXT
                0000A024 DEFEXT  EQU     $A024          DEFAULT EXTENSION
                0000A009 FCSCLS  EQU     $A009          CLOSE ALL FILES
                0000A00F FDELET  EQU     $A00F          DELETE A FILE
                0000A006 FOPENW  EQU     $A006          OPEN FOR WRITE
                0000A002 FWRITE  EQU     $A002          WRITE A BYTE
                0000A029 GETCH   EQU     $A029          GET CHAR
                0000A023 GETNAM  EQU     $A023          GET FILE NAME
                0000A02C INLINE  EQU     $A02C          INPUT TEXT LINE
                0000A037 PERROR  EQU     $A037          PRINT ERROR MSG
                0000A035 PSTRNG  EQU     $A035          CR/LF AND STRING
                0000A000 VPOINT  EQU     $A000          PT TO VAR AREA
                0000A01E WARMST  EQU     $A01E          WARM START
                        *
000000                          ORG      $0000
                        *
000000 6002      )000004 BUILD   BRA.S    START
                        *
000002 0100             DC.W     $0100          VERSION
                        *
                        * ACTUAL START OF PROGRAM
000004 A000             START   DC       VPOINT         GET POINTER
000006 204E                     MOVE.L   A6,A0          SAVE POINTER
000008 284E                     MOVE.L   A6,A4          POINT TO SYS FCB
00000A A023                     DC       GETNAM         GET FILE SPEC
00000C 652E      )00003C        BCS.S    ERROR          ON ERROR
                        *
                        * IF NAME WAS OK, DO DEFAULT EXTENSION
00000E 183C 0001                MOVE.B   #1,D4          CODE FOR DEFAULT
000012 A024                     DC       DEFEXT         GO DEFAULT IT
                        *
                        * NOW OPEN FILE FOR WRITE
000014 A006             OPEN    DC       FOPENW         OPEN FILE
000016 662A      )000042        BNE.S    OPENNG         IF ERROR
                        *
000018 49FA 005C)000076 NXTLIN  LEA      PROMPT(PC),A4  PRINT PERIOD
00001C A035                     DC       PSTRNG         PRINT IT
00001E A02C                     DC       INLINE         INPUT A LINE
000020 2668 02F6                MOVE.L   LPOINT(A0),A3  POINT TO NEXT
000024 181B                     MOVE.B   (A3)+,D4       GET CHAR
000026 0C04 0023                CMP.B    #$23,D4        CHECK FOR #
```

```
00002A 6712      }00003E          BEQ.S    QUIT            YES, SO GO QUIT
00002C 2848                       MOVE.L   A0,A4           POINT TO FCB
00002E A002              NEXTL1   DC       FWRITE          GO OUTPUT CHAR
000030 660A      }00003C          BNE.S    ERROR           ON ERROR
000032 0C04 000D                  CMP.B    #$0D,D4         END OF LINE?
000036 67E0      }000018          BEQ.S    NXTLIN          YES, START LINE
000038 181B                       MOVE.B   (A3)+,D4        GET CHAR
00003A 60F2      }00002E          BRA.S    NEXTL1          AND REPEAT
                          *
                          * PROCESS ERRORS
00003C A037              ERROR    DC       PERROR
00003E A009              QUIT     DC       FCSCLS
000040 A01E                       DC       WARMST
                          *
                          * ERROR HANDLING ON OPENING FILE
000042 0C2C 0003 0001    OPENNG   CMP.B    #3,FCBERR(A4)   ALREADY EXISTS?
000048 66F2      }00003C          BNE.S    ERROR           NO, REAL ERROR
                          *
                          * IF FILE EXISTS, DELETE IT?
00004A 49FA 002C}000078           LEA      ASKDEL(PC),A4
00004E A035                       DC       PSTRNG          ASK TO DELETE
000050 A029                       DC       GETCH           GET ANSWER
000052 0205 00DF                  AND.B    #$DF,D5         CVT TO UPPER CASE
000056 0C05 0059                  CMP.B    #$59,D5         IS IT YES?
00005A 66E2      }00003E          BNE.S    QUIT            QUIT IF NOT
00005C 49FA 0052}0000B0           LEA      ASKSUR(PC),A4
000060 A035                       DC       PSTRNG          ASK IF HE'S SURE
000062 A029                       DC       GETCH           GET ANSWER
000064 0205 00DF                  AND.B    #$DF,D5         CVT TO UPPER CASE
000068 0C05 0059                  CMP.B    #$59,D5         IS IT YES?
00006C 66D0      }00003E          BNE.S    QUIT            QUIT IF NOT
                          * DELETE FILE IF OK WITH USER
00006E 2848                       MOVE.L   A0,A4           POINT TO FCB
000070 A00F                       DC       FDELET          DELETE THE FILE
000072 66C8      }00003C          BNE.S    ERROR           ON ERROR
000074 609E      }000014          BRA      OPEN            OPEN FILE AGAIN
                          *
                          * TEXT STRINGS
000076 2E                 PROMPT  DC.B     "."
000077 04                         DC.B     4
000078 5448 4154 2046     ASKDEL  DC.B     "THAT FILE ALREADY EXISTS ...
                                                DO YOU WISH TO DELETE IT? "
0000AF 04                         DC.B     4
0000B0 4152 4520 594F     ASKSUR  DC.B     "ARE YOU SURE YOU REALLY WANT TO? "
0000D1 04                         DC.B     4
                          *
                                  END      BUILD
```

Although it is not immediately obvious from the above examples, all user-written programs must be written in position - independent code (although see the description of the binary file format in Chapter 13 for possible exceptions.) To write position - independent programs for 68xxx processors, generally follow the following rules:

1. Do not use JMP and JSR instructions - use BRA and BSR instead. In general, there should be no JMPs or JSRs in your programs at all.

2. Refer to variables within your program using PC-relative addressing. For example, the instruction MOVE.B NUMBER(PC),D4 would move the quantity NUMBER into D4, but the (PC) tells the assembler to use PC-relative addressing. Unfortunately, the 68xxx does not allow PC-relative addressing as a destination; that is, the instruction MOVE.B D4,NUMBER(PC) is illegal. Hence this instruction has to be replaced by a two-instruction sequence such as

```
LEA NUMBER(PC),A5
MOVE.B D4,(A5)
```

While this adds an extra instruction every time you store to a local variable, we suggest that you avoid the alternative shortcut of setting one address register to point to your data area and then doing all stores relative to that register. Although this makes your program a bit shorter and faster, it generates code which causes PICTEST to signal an error even though there is none. (PICTEST, explained later in the manual, is used to check a program to make sure it is position independent.)

NOTE to non - assembly language users:

The assembly language examples in this chapter are intended only as guides for those users who intend to write their own assembly language programs. If you wish to try them out, proceed as follows: (1) Type the command EDLIN SAMPLE to edit a sample file. (2) When EDLIN returns with a # prompt, give it the I command to start inserting text. (3) Enter the LIST program (the first program in this chapter). Examine the listing to note that the first line of the program begins with an asterisk; type in only the material to the right of that column. For example, begin the first line with * LIST ...; begin the 7th line with FCBERR EQU ... (4) When you finish typing in the program, enter a # at the beginning of a new line, and then use the S command to exit EDLIN. (5) Give the command ASM SAMPLE to assemble the sample from assembly language to machine language. You will now see that the assembler added all of the machine language code which you did not type in. (6) Assuming there are no errors (correct the program if there are), then give the command SAMPLE SAMPLE. The computer will then use the SAMPLE.COM file generated by the assembler to print out the SAMPLE.TXT file you typed. SAMPLE.COM does exactly the same thing as the LIST command supplied with SK*DOS, except that it does not have the 'help' feature.

13. INFORMATION FOR ADVANCED PROGRAMMERS

This chapter gives additional information for systems or advanced programmers. It describes the disk format, structure of files, and information regarding customization.

DISK FORMAT

A typical disk, whether hard or floppy, is divided into tracks; each track is then divided into sectors. The number of tracks and sectors on a disk depends on the type of disk and drive - a 5-1/4" floppy disk might have as few as 35 tracks with 10 sectors per track, or a Winchester hard disk might have as many as 256 tracks with 32 or more sectors per track. In addition, the disk drive might be able to use both sides of a disk, or a Winchester disk might have multiple disks spinning on the same shaft.

As far as SK*DOS is concerned, the exact number of sides, tracks and sectors is unimportant as long as there are at most 256 logical tracks (numbered 0 through 255) per drive and 256 logical sectors (numbered 0 through 255) per track. (For compatibility with 6809 SK*DOS, sector numbering begins with 1 for floppy disks.)

The exact positioning of those sectors and tracks is controlled by the disk drivers and FORMAT routine, not by SK*DOS itself. On floppy disks, the physical placement of these tracks and sectors on the disk would most likely agree with their logical numbering; on hard disks they might physically be placed elsewhere on the disk. That is why the previous paragraph uses the word *logical* in describing track and sector number - a *logical* address is where SK*DOS thinks the sector is located; the *physical* address is the actual location on the disk where the disk drivers place it.

Depending on the system, SK*DOS floppy disks may be either single- or double density, and single- or double-sided. In addition, double-density disks may have either a single- or double-density directory track. As long as disks are used only on a single system, the particular floppy disk format is not important.

Standard SK*DOS / 68K disks will generally be double density throughout, and may be single or double-sided. Disks intended to be interchanged with 6809 SK*DOS systems, however, should be formatted and used in single-density, single-sided format, since 6809 SK*DOS requires that track 0 always be in single density.

Each sector of an SK*DOS disk contains 256 bytes of data. Of these 256 bytes, the first four are used for system information, and the remaining 252 bytes are usable for file data.

SK*DOS uses a linked-chain disk format. That is, the sectors used in files, as well as sectors which are in the so-called *free chain* are linked to each other much like the links in a chain. Each sector contains a two-byte pointer which points to the next sector in that chain (unless it is 0, which indicates the end of that chain.) This pointer occupies the first two bytes of every sector. In addition, the sector also has a number, which occupies the third and fourth byte, and which counts the sectors within a file.

Thus the sector format looks like this:

Bytes 1 and 2 - pointer to next sector
Bytes 3 and 4 - sector counter
Bytes 5 through 256 - 252 bytes of data

Some sectors have a slightly different format, and may omit the pointer or sector counter.

All the tracks on a disk can be used for storing data and program files except for track 0. The sectors on this track have special uses as follows:

Sector 1 on track 0 holds the super-boot program. This is a program which is loaded by the boot program in the system ROM monitor, and which in turn loads the rest of SK*DOS into memory. (This sector has 256 bytes of data, as the first four bytes of the sector are used for regular data storage rather than being used as pointer and sector count bytes.) On some systems, the boot procedure may be different, and so this sector may not be needed on those systems.

Sector 2 is often empty. It has been set aside as an extension of sector 1 in case more than 256 bytes are needed for booting.

Sector 3 is the System Information Sector or SIS. It contains the disk name and number, the date when the disk was formatted, the number of tracks and sectors on the disk, and three pieces of information about the free sector chain on the disk: the track and sector numbers of the first sector in the chain, the track and sector numbers of the last sector in the chain, and the total number of sectors in the chain.

Sector 4 is usually empty, although the COPY utility places a copy of the SIS into this sector to verify that the disk is available for writing.

Sector 5 begins the directory, which extends to the last sector of track 0. Each directory entry requires 24 bytes, so there is room for 10 entries in each sector with 16 bytes empty. For example, on a 5-14" single density, single sided disk, there are 10 sectors in track 0. Hence there are six sectors in the directory, numbered from 5 to 10, for a total of 60 directory entries. The six sectors are linked (through the first two bytes in each sector, and the last sector has a pointer of 00-00. When the directory is filled up, however, SK*DOS will take a sector from the free chain and add it to the directory, so that the directory can be expanded to make room for more entries (although this may greatly slow down the operation of the system if the added directory sector is on one of the inner tracks of the disk since the disk head will have to step in and out each time it accesses the directory.)

SEQUENTIAL FILES

Most SK*DOS files are of the sequential type (as opposed to random files, discussed next). Sequential files are intended to be read in order, from beginning to end. Such files generally are of two types - text or binary.

Text File Format

SK*DOS text files consist simply of ASCII text, usually separated into lines of text by CR ($0D) characters; LF characters ($0A) are not included. Most text files use space compression, where two or more consecutive spaces are instead replaced by the TAB character ($09), followed by a byte representing a space count between 2 and 127, inclusive. Strings of spaces of length greater than 127 are broken up into smaller pieces, each of length 127 or less.

No special character is used to denote the end of text, although the last line of text will generally end with a CR. Any space remaining in the last sector of a text file is filled with NULL ($00) bytes. When SK*DOS reads a space-compressed file, it does not return any NULL characters to the calling program; hence it will generate an end-of-file error immediately after the last character of the text.

Text files may consist of any characters except for NULL ($00) and TAB ($09).

Binary File Format

SK*DOS binary files are non-space-compressed files which contain binary data along with additional information which specifies where in memory that data is to be loaded and/or executed.

A typical binary file will generally consist of several segments, each of which begins with an identification byte which describes what the segment consists of. There are ten such identification bytes:

$02 marks the beginning of a relatively short segment containing binary data to be loaded into memory. The $02 is followed by a two-byte load address, a one-byte count which specifies how many bytes are to be loaded, and a number of bytes equal to the count. The count is a number between 1 and 255, and the load address is a number between $0000 and $FFFF. During loading, SK*DOS adds the current value of OFFSET to load addresses specified in the file (unless the - option is used in the command line).

$03 is similar to $02 in that it also marks the beginning of data to be loaded into memory, but it is followed by a four-byte load address and a two-byte count. It is therefore used for memory addresses above $FFFF, or for loading data longer than 255 bytes (although such data is often split into a number of shorter $02 segments.) As with the $02 segment, SK*DOS adds the current value of OFFSET to load addresses specified in the file (unless the - option is used in the command line).

$16 marks the beginning of a two-byte transfer address; that is, the address where the file just loaded should be executed. The current value of OFFSET is added to the address specified in the file.

$17 marks the beginning of a four-byte transfer address, used if the transfer address is above $FFFF. The current value of OFFSET is added to the address specified in the file.

$04, $05, $18, and $19 are similar to $02, $03, $16, and $17, respectively, except that the current value of OFFSET is NOT added to the specified address in the file. Note, however, that load addresses are still checked against OFFSET and MEMEND limits unless the GETX command is used (see the descriptions of GET and GETX later in this manual). These four codes are provided for special applications, and should not normally be used as they may cause the system to crash in future multi-tasking versions of SK*DOS.

$0F and $10 are special codes used for programs which are not written in the normal position independent code (PIC). They are used to allow SK*DOS to modify an address while loading a position - dependent program. The $0F is to be followed by a single word address (relative to OFFSET), and the $10 is to be followed by a single long-word address (relative to OFFSET), which specifies the address of a long-word address which is to be modified by adding the current value of OFFSET to it. For example, suppose the current value of OFFSET is $5000 and the disk file contains the sequence $0F 12 34 or the sequence $10 00 00 12 34. Either of these two segments tells SK*DOS to add $5000 to the contents of address $6234 (which is presumed to have been previously loaded.) This code would normally be used only with assemblers or compilers which generate non - position - independent code.

With one exception, all of the loading and transfer addresses referred to above are merely relative addresses; they are added to the current value of OFFSET (see Chapter 11) when used. For example, if OFFSET currently has a value of $5000, and a file has a loading address of $1000 and a transfer address of $1004, then it will actually be loaded into memory at $6000 and executed starting at $6004. The exception is SK*DOS itself. Since SK*DOS is loaded by the bootstrap program at a time when OFFSET has not yet been defined, it contains absolute loading and transfer addresses rather than relative ones. Hence you can determine the absolute address where your SK*DOS is loaded into memory by examining the SK*DOS.SYS (or SK*DOS.COR) file with the

LOCATE command (using the - option so the OFFSET is not added by LOCATE). Since COLDST is located at the very beginning of SK*DOS, this method is also used to find the address of COLDST in your system.

Note that segments do not contain any kind of checksum; it is assumed that any disk errors will be caught by CRC or other error checking in the disk hardware or drivers. As in text files, the last segment of a binary file is generally followed by NULLs so as to fill out the last sector of a file.

RANDOM FILES

The directory for each file only gives the track and sector numbers of the first and last sector of the file; it does not contain any information as to which other sectors the file uses. Since the sectors of a file need not necessarily be consecutive on the disk - they could lie anywhere on the disk if the disk has been much used and its free space is not all in one place - the file itself contains information linking one sector to the next. This is done by the first two bytes in each sector, which contain the track and sector number of the next sector in the file. In a sequential file, we normally start reading or writing at the beginning of a file and then continue through the file, following these two-byte links until we get to the end.

As pointed out earlier, the third and fourth bytes of a data sector contain a two-byte (four hex-digit) *sector count* which numbers the sectors of a file. For example, the first data sector of a file has the number 0001, the second is numbered 0002, and the seventh would be numbered 0007. Don't confuse these numbers with the physical location of the sector on the disk, which is sometimes called the sector's *disk address*, and which consists of a track number and a sector number. The *sector count* could be used to make sure we read the sectors of a file in the right order, but in practice is almost never used with sequential files. Though we have used the term *sector count* above, in the rest of this discussion we will simply call it the *sector number*. Some people also call it the *record number*, but this is a bit confusing since the word *record* can also be applied to a part of a sector.

Although sequential files are most common, we often need *random* or *direct access* files. These are files which allow us to read or write data in the middle of a file without necessarily reading or writing all the data before it. For example, at some point we might need to read data located in sector 0007, followed by sector 0104, followed by sector 0025, and so on. This is accomplished by the random file capabilities of SK*DOS.

In order to allow us to rapidly locate a particular sector number in a file, without reading all the sectors before it, SK*DOS provides for a special random file format which contains an extra two sectors. These two sectors, which are always at the very beginning of a random file, contain a *file map* of the file, which maps out the placement of the file on the disk and helps us to find a specific sector. These two sectors always have a sector number of 0000.

Thus the very first data sector of any file is always sector number 0001, but in a sequential file this is the first actual sector of a file, whereas in a random file it is actually the third sector. (If you try to do a sequential read of a random file, SK*DOS skips the first two sectors and so you will never know they are there.)

Few application programs actually need random files, but if they do, they will take care of generating and manipulating them automatically, through SK*DOS. Nevertheless, here is information on how this is done.

There is only one way to generate a random file:

1. Open a file for writing with FOPENW.
2. After opening, but before writing anything to it, change byte 26 of the FCB to a non-zero number.
3. Now write sequentially to the file using FWRITE.
4. When done, close the file with FCLOSE.

SK*DOS will automatically put a two-sector file map at the beginning of the file, and will update it as data is written to the file. Note that the file map is customized for the particular placement of the file sectors on the disk. If you copy a file from one disk to another, the two file map sectors will be different since the copy will most likely be in a different place on the disk. But you need not concern yourself with this since SK*DOS does this automatically.

Once the file is written, it can be read or updated (the data in it can be modified), or lengthened. But random files are usually made oversize to begin with, so there is room for adding more data at a later time without increasing the file size later.

The file can be read sequentially just like any sequential file. If it is opened (with FOPENR) and then read (with FREAD), it will look like any sequential file, since SK*DOS will automatically skip the file map sectors and read only the data sectors.

Things are a bit more complex - and interesting - if the file is opened for updating. We now have a number of options, which are listed in the description of the Open for Update operation, FOPENU. The most important concept is that, by use of FRRECD, we can locate any sector number (that is, a sector with a desired sector count) in the file in a short time. For example, we can tell SK*DOS to read sector number 0104 of a file. Given a sector number, SK*DOS will look it up in the file map, calculate the exact location of that sector, and then go directly to it.

Once we have located the specific sector, we can specify an exact byte of that sector, and either read it (with FRGET) or write it (with FRPUT). Knowing that there are 252 data bytes in each sector, we could thus locate any particular byte in a file after some fairly simple calculations. For example, to locate byte N in a file, we could use the following two BASIC lines:

```
Sector number  =  INT(N/252) + 1
Byte number    =  N - INT(N/252)*252 + 4
```

N in these equations is assumed to start with 0; the 4 in the second equation is due to the fact that the first data byte in a sector is actually numbered 4. For example, the 253rd byte in a file (which would actually be numbered 252) would be byte number 4 (the first byte) in sector 0002.

OTHER USEFUL ADDRESSES

There are some additional addresses which will be of interest only to programmers who wish to customize SK*DOS for their specific systems. These are located at fixed offsets above COLDST; see the SEQUENTIAL FILES part of this chapter for a description of how to find COLDST.

Warmstart COLDST+$06 Warm Start SK*DOS

If you exit SK*DOS to a ROM monitor or other debugging tool and wish to return, you may do a jump to this Warmstart location. (Don't confuse this location with the WARMST trap described in Chapter 10.)

GETDAT COLDST+$0C Get boot date

At GETDAT there is a JMP instruction which points to the routine which asks you for the date when booting SK*DOS. If there is a clock/calendar IC in your system, you may replace this JMP with a call to your own routine which reads the date from this IC. You must preserve all registers.

INTIME COLDST+$12 Add time to directory entry

INTIME normally contains three RTS instructions. Each time that SK*DOS opens a file for writing, it places the next sequence number for the file into D5, does a JSR to the INTIME trap (with an immediate RTS because INTIME normally contains six bytes of RTS), and upon return stores the contents of D5 into byte 27 of the current FCB. D5 normally contains the sequence number, but a user may replace the RTS bytes with a jump into a user-written routine which may replace the byte in D5 with a time of day byte. One byte is not enough to indicate a precise time down to the minute, but it can be used to represent time in tenths of hours, resulting in 6-minute resolution. A byte of 00 means no sequence number or time is present; 01 is a time from 12:00 midnight to 12:05 a.m.; 02 is 12:06 a.m. through 12:11 a.m., and so on. (User programs using INTIME must preserve all registers except for D5.)

OFFINI COLDST+$18 Initial OFFSET value

OFFINI generally points to an even location just above SK*DOS; on each cold-start or warm-start, SK*DOS reads OFFINI and then initializes OFFSET at the next even 256 byte address above OFFINI. (For example, if OFFINI is $49EA, then OFFSET will become $4A00.) See the next section for a discussion on how to use OFFINI, OFFSET, and MEMEND to reserve space for custom routines.

MEMINI COLDST+$1C Initial MEMEND value

When SK*DOS is initially booted, it does a memory sense to determine how much memory is installed in the system; it then uses this information to set MEMEND. MEMINI sets the highest address that SK*DOS will try during that test. This is essential in those systems which may not return a buss error if non-existent addresses are accessed, but can also be used to set aside an area of memory which SK*DOS will never use.

SECTRD COLDST+$20 Secondary sector read routine trap

SECTRD normally contains 10 RTS instructions (a total of 20 bytes) which are called each time SK*DOS calls the SREAD routine to read a sector. A4 at this time points to the FCB being used for the read. A user may substitute up to three alternate sector read routines by inserting JSR instructions which point to these other routines. This allows the simple addition of a cache, RAM disk, or alternate disk controllers. Such routines must preserve A0 through A4 and D0 through D4. When such JSR instructions are added, they will normally go to a routine which analyzes the drive number in the specified FCB. If the drive number is different from the one handled by this secondary disk read routine, then it should perform an RTS to return to SK*DOS; if the drive number matches, then the routine should perform the requested read, remove from the stack the extra return address placed there by the added JSR, and then RTS directly back to the calling program.

SECTWR COLDST+$34 Secondary sector write routine trap

SECTWR normally contains 10 RTS instructions (a total of 20 bytes) which are called each time SK*DOS executes the SWRITE routine to write a sector. A4 at this time points to the FCB being used for the write. A user may substitute up to three alternate sector write routines by inserting JSR instructions which point to these other routines. This allows the simple addition of a cache, RAM disk, or alternate disk controllers. The program must preserve A0 through A4 and D0 through D4.

SECCOL COLDST+$48 Secondary cold-start initialization

SECCOL normally contains 10 RTS instructions (a total of 20 bytes) which are called during cold start. This area would normally be used for initializing any drivers used with SECTRD or SECTWR. A user may call up to three initialization routines by inserting JSR instructions which point to these other routines. j The program must preserve A0 through A4 and D0 through D4.

SECWAR COLDST+$5C Secondary warm-start initialization

SECWAR normally contains 10 RTS instructions (a total of 20 bytes) which are called during warm start. This area would normally be used for initializing any drivers used with SECTRD or SECTWR. A user may call up to three initialization routines by inserting JSR instructions which point to these other routines. The program must preserve A0 through A4 and D0 through D4.

SECCHK COLDST+$70 Secondary disk ready check

SECCHK normally contains 10 RTS instructions (a total of 20 bytes) which are called to check whether the secondary disk drivers are ready when SK*DOS is searching for the next ready drive. At that time, A4 points to an FCB which contains the drive number, and the drive number is also in D5. The called routine should return a *zero* condition if the requested drive is ready, or *non-zero* if it is not ready. The program must preserve A0 through A4 and D0 through D5.

SECFL1 COLDST+$84 Secondary Flag 1

SECFL2 COLDST+$8A Secondary Flag 2

SECFL3 COLDST+$90 Secondary Flag 3

These are three six-byte areas which may be used by secondary drivers as general purpose flags.

TRPFLG COLDST+$C0 Trap Initialization Flag

TRPFLG determines whether SK*DOS will initialize only the "Line 1010" trap (if TRPFLG = 0) or all the traps (if TRPFLG is non-zero) each time that RESTRP is called. This decision is largely based on the type of system, and should not normally be changed by the user. If TRPFLG is zero, then traps (such as bus error etc.) will be handled by the monitor; if TRPFLG is non-zero, then traps will be handled by SK*DOS.

RESERVING MEMORY WITH OFFSET AND MEMEND

Since OFFSET and MEMEND point to the beginning and end, respectively, of free user memory, they can be used to set aside memory for other programs which should co-exist with SK*DOS without interfering with it.

When SK*DOS is initially booted, it is loaded with initial values of OFFINI and MEMINI. It first checks memory, starting with OFFINI up through the value of MEMINI, looking for a memory address which fails to store a memory pattern stored into it, or which generates a bus error. Once such an address is found, SK*DOS stores into MEMEND the top address of the last 4K block of memory found to be working. This value of MEMEND then remains there unless changed by user programs; SK*DOS does not itself change it at any time (although programs such as RAMDISK may.)

Following the above, at every warm-start SK*DOS sets OFFSET to the next even 256-byte boundary above the current OFFINI. Since OFFINI at boot-up normally points just above SK*DOS and its drivers, OFFSET is thus initialized to the next 256-byte boundary above SK*DOS.

PROCESSOR STATE AND STACK USE

Since user programs call SK*DOS with traps, SK*DOS obviously runs in supervisor state. User programs, utilities, and application programs, however, run in user state.

The system stack pointer is initialized at COLDST, with the user stack pointer initialized at COLDST-$0200. For example, if COLDST is at $1000, then the user stack will go from $0E00 down, while the system (DOS) stack will go from $1000 down. The system stack will never extend down as far as the user stack, so data on the user stack is preserved during DOS calls. Between them, OFFSET (through OFFINI) and MEMEND (through MEMINI) therefore delimit the current user memory.

User programs (and SK*DOS utilities) can subsequently change these values to reserve memory for themselves. A program which wants to reserve permanent memory for itself (such as a RAM disk) can be loaded into memory at OFFSET, and then point OFFINI above itself, or can move itself to the top of memory and then point MEMEND just below itself. Either way, its memory will then be protected permanently, since SK*DOS does not itself move these boundaries.

It is also possible to reserve memory just temporarily; for example, a program might be needed only for a while. Such a program can then be loaded into low memory, and OFFSET adjusted to point to an even address above the program. OFFSET will then remain at this value only until the next warm start, at which time it will go back to the previous value as determined by OFFINI. (Note that OFFSET must always be even.)

USER-INSTALLED MEMORY-RESIDENT COMMANDS

Users may add their own memory-resident commands to the list normally contained within SK*DOS. The program code for such commands may be left in memory and protected from SK*DOS in one of several ways:

1. The program can be placed at the bottom of user memory and then OFFINI set to point above the program's top memory address, or

2. The program can be placed at the top of user memory and then MEMEND set to point just below the program's bottom address, or

3. The program can be placed into some other memory area which SK*DOS does not use and does not know about.

To add the command to SK*DOS's command list, a table of command names and starting addresses must be placed somewhere in memory and identified to SK*DOS. The following is an example of such a table:

```
CMDTAB DC.B 'DIR' Command name
       DC.B  0    delimiter to signal end of name
       DC.L  DIR  Starting address of command
       DC.B 'GET' Command name
       DC.B  0    delimiter to signal end of name
       DC.L  GET  Starting address of command
       .   .
       .   .
       .   .
       DC.B  0    End of table flag
```

Finally, to tell SK*DOS where this table is located, its starting address (i.e., the address of the first DC.B) must be placed as a long word into location COMTAB.

(The astute reader may note that, depending on the length of the command name, the assembler may insert an extra empty byte between the DC.B command name delimiter and the DC.L which holds its address so as to make sure that the DC.L starts on an even address. This is irrelevant to SK*DOS.)

TRACING PROGRAMS

If you have a ROM system monitor, such as HUMBUG, which supports tracing programs, you may use SK*DOS's TRACE*** command to enter a program in the trace mode (TRACE*** has three asterisks so that you will not accidentally type in the command when you don't mean to.)

TRACE*** is generally used just before you load in and execute a new program being tested. It sets the trace bit in the user status register, so that SK*DOS enters the new program in the trace mode. The 68xxx CPU will then execute the first instruction and trap to the ROM monitor, generally to display a register dump. You may then use the facilities of the monitor to trace further, insert breakpoints, or do other debugging.

When using TRACE*** with HUMBUG, you must first execute the TRACENAB command, which tells HUMBUG that you will be using TRACE*** and initializes one of its memory locations.

SYSTEM INFORMATION SECTOR FORMAT

The System Information Sector (SIS) contains the following data; all unused bytes are 00.

Bytes 16-26	Disk name (and extension)
Bytes 29-30	Track and sector number of first free sector
Bytes 31-32	Track and sector number of last free sector
Bytes 33-34	Number of free sectors
Bytes 35-37	Month, day, and year of disk creation
Byte 38	Number of logical tracks on the disk - 1

Byte 39 Number of logical sectors per track

DIRECTORY STRUCTURE

The directory of a disk occupies track 0 of every disk, beginning at sector 5 and extending to the end of the track. This area of a disk is reserved for the directory when the disk is initially formatted, but SK*DOS will extend the directory, one sector at a time, if additional space is needed. Like any other file, the directory is a linked chain; if additional sectors are needed, they may be anywhere on the disk.

Within each sector, the first two bytes are a link pointer and the next 14 bytes are empty (filled with zeroes); the remaining 240 bytes are split into ten groups of 24, with each set of 24 bytes being one file entry. These bytes are used as follows:

Bytes 0-10	File name and extension
Byte 11	File attribute
Byte 12	File protection
Bytes 13-14	Track and sector number of first sector
Bytes 15-16	Track and sector number of last sector
Bytes 17-18	File size in sectors
Byte 19	Sequential/random flag
Byte 20	Time of file creation / update
Bytes 21-23	Month, day, and year of file creation / update

The first character of the file name (byte 0) is replaced by $FF when a file is deleted, but the remaining bytes are unaltered. Hence the CAT command (using its N option) can display data on deleted files. When the directory is first established, all bytes in the empty directory are written as zeroes; hence when SK*DOS searches the directory for a file, it stops searching when it gets to an entry whose first character is still 00.

Although logically an SK*DOS disk can hold one root directory and 26 (or more) subdirectories, physically all of the files are listed in one master directory, which is stored as a flat file as described above. Within this directory, files are coded as to which subdirectory (A/ through Z/) they belong into by storing the directory code in bit 7 of bytes 0 through 7 of the directory entry.

As with all ASCII text in SK*DOS, file names are stored as 7-bit ASCII characters, with the left-most, eighth or parity bit, normally a 0. Bit 7 of bytes 0-7 (the eight bytes of the file name) would therefore normally be zeroes. Instead, they are now used to hold the directory code letter. For example, here are some sample file-name bytes:

File TEXT in the root directory:

```
54  45  58  54  00  00  00  00  <- ASCII for "TEXT" plus four nulls
0   0   0   0   0   0   0   0   <- parity bits; 00 means root
```

File TEXT in directory U/:

```
54  C5  58  D4  00  80  00  80  <- ASCII for "TEXT" plus four nulls
0   1   0   1   0   1   0   1   <- parity bits; $55 means "U"
```

While at first glance this seems like an awkward way of coding subdirectories, in practice just a few extra instructions are required to process these parity bits. SK*DOS routines GETNAM and FNPRNT do the processing automatically for input and output of file specifications. The advantages, on the other hand, greatly outweigh the disadvantages: the directory structure is totally compatible with earlier versions of SK*DOS, same routines which differentiate between file names also diferentiate between directory names without any extra

programming or time, it becomes easy to move a file from one directory to another without rewriting it, the directory remains a manageable size, and for most situations, the entire directory for an entire disk is still contained in just one track, thereby minimizing disk access time.

14. I/O REDIRECTION AND I/O DEVICES

This chapter gives additional information on the entire interrelated (and inter-twined) subject of I/O redirection, device drivers, printers, communications, and the like. It supplements information given for the DEVICE command in Appendix G. In particular, it describes the differences between 'device names', 'device numbers', and 'device drivers'.

Before continuing, it is important to explain our use of the word *device*. In SK*DOS, a device is any I/O port other than a disk drive or RAMDISK. This includes character-oriented devices such as the console, terminals, modems, or printers. For the sake of the discussion in this chapter, we will differentiate between disks and devices, even though many people would classify a disk as one type of device. In fact, SK*DOS allows devices and disks to be treated in similar ways. Nevertheless, for our purposes it is easier to separate the concepts.

COMMAND LINE REDIRECTION

In its simplest form, I/O redirection simply means sending output to a different place, or accepting input from a different place, than normal. It can be accomplished directly from the command line by using the symbols > and <. For example, the command

```
CAT
```

normally displays a catalog of a disk on the screen. On the other hand, the command

```
CAT >CATFILE
```

sends the catalog listing to a diskfile called CATFILE.PIP, rather than displaying it on the screen. Furthermore, the command

```
CAT >PRTR
```

would send the catalog listing to the printer (if a PRTR printer driver is installed; otherwise it will go to a disk file called PRTR.PIP. You can also force output to a PRTR file, even when a PRTR device is installed, simply by including the extension.)

Input redirection is handled with the < symbol. For example, the command

```
BUILD FILE
```

is generally used to input text from the keyboard into FILE. The command

```
BUILD FILE <ANOTHER
```

would also send text to FILE, but would take the text from another file called ANOTHER.PIP. Alternatively. if there is a device called COM1 on the system, then

```
BUILD FILE <COM1
```

would take the text from this input device.

Think of < and > as being arrows. The > in >FILE points to FILE, so data goes to FILE, whereas <FILE points away from FILE so data comes from FILE. Note also that file names used in redirection default to .PIP extensions (which stands for 'pipe'), though this can easily be changed by specifying a different extension.

DEVICE NAMES AND DEFAULTS

In the above examples, PRTR and COM1 were 'device names', as opposed to 'file names'. Device names are similar to file names, but (a) must have exactly four characters, and (b) are not allowed any extensions. Whenever you use a device name, SK*DOS checks whether such a device exists. If it does, then it uses the device. If not, then it uses the same name as a file name. That's why in the above example >PRTR went to a printer if such a device existed, but to a disk file otherwise.

When SK*DOS is initially booted, it has just two devices; these are called the 'default' devices:

CONS is the console keyboard and screen. It is used for normal input and output.

NULL is a 'null device' which is used when you want to do a function but want no output whatsoever. For example, the command

```
ASM PROG >NULL
```

would assemble a file but provide absolutely no output - not even assembler error messages.

Even though it starts with just two devices, SK*DOS can have up to eight. The others must, however, be specifically 'installed', either by a command from the keyboard, or by commands included in a STARTUP.BAT file, by reading in 'device drivers' from a disk. If you wish, you can substitute other devices instead of the default ones as well.

DEVICE NUMBERS

In addition to having names, devices also have device numbers. When using I/o redirection, you will always refer to them by name; programs, on the other hand, may refer to them by either name or number. The DEVICE command lets you see (and change) the correspondence between device names and numbers. If you execute the DEVICE command just after booting up SK*DOS, you will get a display like this:

Normal use	Device number	Device name	Driver
Terminal	0	CONS	Default driver
Error device	1	CONS	Default driver
Printer	2	CONS	Default driver
	3	CONS	Default driver
	4	CONS	Default driver
	5	CONS	Default driver
	6	CONS	Default driver
Null device	7	NULL	Default driver

This tells us that devices 0 through 6 are currently the CONS console default driver, while device 7 is the NULL default driver. Furthermore, it also tells us that device 0 is normally the terminal (which is used to control

SK*DOS), device 1 is the error device (where most error messages go), device 2 is usually the printer (although right now printer output would go to the console instead), and device 7 is usually the null device.

As you can see, at this point CONS has several device numbers. That means that output sent to any one of those numbers would really go to the console.

The assignments shown in the above tables can be changed at any time by using the DEVICE command. DEVICE is most often used to substitute a disk-resident device driver for one of the default drivers.

DEVICE DRIVERS

In most systems there will be just one console, but there could be several printers. Moreover, a printer could require either a serial interface or a parallel interface. Hence the software to drive a printer, as well as other devices, must be changeable so it can fit the hardware. This is done by using disk-resident programs to interface with other devices. These programs are called 'device drivers', and usually have a .DVR extension on the disk. Depending on your system, you may already have one or more such drivers supplied with your SK*DOS, or else you may just have one or more files of driver source code which you will have to customize and assemble to fit your own hardware.

Device drivers must be 'installed' with the DEVICE command. For example, the command

```
DEVICE PARALLEL AT 2 AS PRTR
```

would install a driver called PARALLEL.DVR at device 2, and give it the name PRTR. The DEVICE display would then say

```
  Normal      Device   Device    Driver
   use        number    name
  ------      ------   ------  ---------
    .           .        .         .
 Printer        2       PRTR    PARALLEL.DVR
    .           .        .         .
```

The DEVICE command can be used to change devices as often as desired; you may also return back to a default driver by using the name DEFAULT instead of a file name, as in

```
DEVICE DEFAULT AT 2 AS CONS
```

which would restore device 2 as the default device CONS. Note that driver names default to .DVR, but DEFAULT has no default extension: DEFAULT refers to the normal default driver, whereas DEFAULT.DVR would be needed in the command line if you had an actual DEFAULT.DVR driver on a disk.

REMOTE CONSOLE OPERATION

It is possible to operate SK*DOS from a device other than the default console device. For example, if there is a serial device on the system and a driver for it, then you may install that driver as devices 0 and 1. All normal console I/O would then go to that device instead. Note that it's necessary to install that driver at both number 0 and 1 so that error messages go to the new device. The order of assigning device numbers also makes a

difference - if you assign number 0 first, then you will have to use the remote keyboard to assign number 1, as the console keyboard will no longer be active.

NOTE: The remainder of this chapter will probably only be of interest to advanced programmers.

DRIVER MEMORY ASSIGNMENT

When a new driver is loaded from disk, DEVICE checks to see whether the driver is smaller than the driver currently installed under that device number. If so, then the new driver simply overlays the current driver. If not, then the new driver has new memory assigned to it just above the current value of OFFSET, and then OFFSET and OFFINI are moved up above the new driver. (The default CONS and NULL drivers have zero size, and so will never be overlaid.) If the same driver is used under various device numbers, several copies of the driver will exist in memory at the same time, one for each device number.

This is important to remember for several reasons. First, it means that new drivers can only be installed from the keyboard (or from a .BAT file), not from another program, because memory may not be available for the new driver while another program is running. Second, it means that each copy of a driver maintains its own variables such as PLINES (see below) even when it applies to the same hardware device.

DEVICE DESCRIPTOR TABLE

Information on device assignments is stored in the Device Descriptor Table called DEVTAB. This table consists of 640 bytes, (80 bytes for each of the 8 device numbers) plus an additional 80 bytes for the default CONS driver. The CONS information is copied into the rest of DEVTAB during booting; the DEVICE command then modifies the contents of DEVTAB when it installs other drivers.

Each device number has an 80-byte device descriptor within the table. These bytes contain the following:

```
 Bytes    Description
00-03 Logical name, such as CONS or PRTR
04-07 Pointer to the first address of the driver, 0000 if in BIOS
08-11 Length of the driver in bytes, 0000 if in BIOS
12-15 Pointer to driver initialization routine
16-19 Pointer to input status check routine in driver
20-23 Pointer to get input character with echo routine
24-27 Pointer to get input character without echo routine
28-31 Pointer to input channel control routine (for ICNTRL)
32-35 Pointer to output status check routine
36-39 Pointer to output character routine
40-43 Pointer to output channel control routine (for OCNTRL)
  44  Print lines (PLINES) constant
  45  Page width constant (PWIDTH)
  46  Null wait constant (NULLWT)
  47  Skip lines constant (SLINES)
  48  Pause flag (PAUSEB)
  49  Line counter counts lines per page
  50  Column counter (OCOLUM)
  51  Serial device baud rate (BAUDRT)
  52  End-of-file character (EOFILC)
```

```
   53  X-Off character (XOFFC)
   54  X-On character (XONC)
   55  Reserved for future use
56-59  Pointer to input status check routine (bypass typeahead)
60-63  Pointer to get input char w/o echo routine (bypass typeahead)
64-67  Pointer to routine to flush typeahead buffer
68-79  Reserved for future use
```

All of the above pointers and numbers are distinct for each device number. The constants from PLINES down can be displayed or changed with the DOSPARAM command.

DEVIN, DEVOUT, AND DEVERR

DEVIN, DEVOUT, and DEVERR are three bytes which indicate the current input, output, and error device number, respectively. Normally, DEVIN and DEVOUT contain the number 0, indicating that they use device 0, while DEVERR contains the number 1, indicating that error messages from PERROR normally go to device 1.

I/O REDIRECTION FROM PROGRAMS

Temporary I/O redirection from the keyboard can only be done one way - by using the > and < symbols on the command line. Permanent redirection can be done by installing another driver.

I/O redirection is done by programs in a totally different way (since programs cannot install new drivers). In general, there are several methods available to programs for accessing different I/O devices. Moreover, programs can access devices through file control blocks, or can access files through device numbers.

1. Normal console I/O functions such as PUTCH, GETCH, PSTRNG, INLINE, OUT5D, HEXIN, and the like, are all steered through DEVIN in the case of input functions, or DEVOUT in the case of output functions. Programs can change DEVIN, or DEVOUT to different device numbers to use different devices for these functions. For example, when a program wants to output to a printer, it can change DEVOUT from 0 to 2 - assuming that a PRTR or similar driver is installed. If not, then output will still go to the console. It is also possible to input from one device but output to another by changing DEVIN and DEVOUT accordingly.

2. An indirect way of changing DEVIN and DEVOUT is through ICNTRL and OCNTRL calls $FFFx (see the next section.)

3. PERROR output is done via DEVERR, and programs can change this byte to steer error output to different devices.

4. All of the above can also be sent to a disk file, or input from a disk file, by opening the appropriate file and placing the FCB address into FIADDR (for input) or FOADDR (for output), and then setting DEVIN or DEVOUT to device number 8. Note that physical I/O devices are only numbered 0 through 7; number 8 applies only to disk files. If the file is not open, or if SK*DOS encounters an error while using the file, it will print an error message, reset the device number to 0, and continue using device 0 for input or output. (You may then use the EOFILC, usually control-Z, to indicate an end of file on input.)

5. Conversely, an FCB can be used to access an I/O device simply by using the four-letter device name when opening the file. Just be careful not to try to input from an output device such as a printer.

The process would go like this: First place a four-letter device name into the name bytes of the FCB. Do not use an extension or SK*DOS will assume you mean a file, although you should call DEFEXT to put in a default extension just in case (DEFEXT will not add an extension if it detects that a device exists with the specified name.) Then call FOPENR or FOPENW to open the file for reading or writing. If the specified driver does not exist, SK*DOS will open the file normally (that's why it is good to have a default extension). If it does exist, then SK*DOS will use the device for subsequent reads or writes instead of a file. Note that the device can only be used sequentially - random file operations will not work and may give undesired results.

SK*DOS accesses devices through an FCB by substituting 'fake drive numbers'. Normally, only drive numbers 0 through 9 are valid disk drive numbers; when you open an FCB to a device, SK*DOS uses drive numbers $10 through $17 to refer to devices 0 through 7 respectively. If you know the name of a device but not its number, then it is easiest to open the file with that name. If you already know the number, then it is not even necessary to open the file - just set up an FCB, put in a drive number equal to $10 plus the desired device number, and use FREAD or FWRITE to read or write.

6. Before doing any of the above, a program may check whether a given driver is installed by using FINDEV.

7. GETNAM recognizes device names and substitutes the device number plus $10 when the specified device is installed.

ICNTRL and OCNTRL

ICNTRL and OCNTRL are two SK*DOS system calls which pass data and commands to and from device drivers without going through the normal GETCH and PUTCH calls. For example, when a user program calls ICNTRL with the instructions

```
MOVE.B  #$10,D4
DC      ICNTRL
```

the value of $10 is passed through SK*DOS to the ICNTRL entry point of the appropriate driver.

The need for ICNTRL and OCNTRL is based on the need for consistency when SK*DOS is implemented on a variety of very different computers. Some of these use conventional terminals, but some (such as the Atari or Amiga 68000 computers) will have built-in video and graphics interfaces. Since each of these provides different output modes and screen display codes, it is important to standardize input and output so that a given program may run on any of these and still provide a common output format.

All 68K SK*DOS calls to GETCH and PUTCH (as well as related calls such as PNSTRN or INLINE) are sent to the specified driver routine through a portion of SK*DOS called IOSEL or I/O Selector. As explained above, the usual device assignments are

```
0 - console (both keyboard and screen)
1 - error device (usually also the console)
2 - printer
3-6 - user-defined
7 - the 'null' device
```

One function of ICNTRL and OCNTRL is to choose which driver is active at any time. When you call ICNTRL or OCNTRL with the word $FFFx in D4, this selects driver x for input or output, respectively. All following I/O calls via PUTCH, GETCH, etc., (as well as ICNTRL or OCNTRL) are then vectored to that driver until you

change the driver assignment with another call to ICNTRL or OCNTRL (or changed DEVIN or DEVOUT). On initial cold start, 68K SK*DOS initializes both input and output to $FFF0 so that the system defaults to using device 0 - the console keyboard and screen. Hence the casual user need not generally be concerned with ICNTRL or OCNTRL.

The second function of ICNTRL and OCNTRL is to pass special arguments to the selected device, or input special key characters from the device. The important requirement is that all SK*DOS users agree on these arguments, so that all drivers and all I/O devices will respond in the same way, regardless of which computer is being used. The following sections describe this feature.

ICNTRL Assignments

ICNTRL is used by placing a command code into D4, and doing a DC ICNTRL instruction. The device driver may then return an argument in register D5. The current command codes are as follows:

$0000	Return number of current driver (0-7) in D5
0001	Return the name of current driver (such as CONS) in D5
0002	Return a raw 8-bit character from the device
0003	Enable keyboard's function keys
0004	Disable keyboard's function keys
0005	Return a special character from the device
FFFx	Switch to driver x

Additional other commands may be defined in the future.

Some device drivers will contain an input translation table and code which allows the driver to convert special key characters or sequences into a single byte which will be returned in D5. For example, suppose a given terminal has a row of ten function keys, which generate a two-byte sequence such as "ESCape followed by $30" for key F0, and so on. Using GETCH, we would get back two separate characters, a $1B for the ESCape, and a $30 for the 0. The problem here is that we have no way of knowing whether this sequence came from function key F0, or whether the user really typed an ESCape and then the digit 0. (The command 0002 of ICNTRL would return the same two characters, but with the parity bits intact, if any.)

ICNTRL command 0005 works a bit differently. After it detects the ESCape code, ICNTRL waits for approximately one-half character time. If it receives the $30 during that time, then it assumes that the combination came from a single F0 function key, and returns a special code which signifies the F0 key. If the $30 is not received during that time, then ICNTRL returns the ESCape first, and then picks up the next character on the next pass. (The next character may also be returned by GETCH). (This description assumes a serial terminal keyboard; computers with an integral keyboard may return a special function key code directly.)

In order to generate the same special F0 code with different terminals or computers, the device driver has a translation table which converts any specific combination of one or more keys into an F0 code which would be common to all systems. The supplied ADM-3A driver shows how this is done.

When ICNTRL command 0005 receives a regular ASCII character, it simply returns it as a single byte in D5, with bits 8-15 of D5 equal to 0. But a special keyboard character is identified by making bits 8-15 non-zero; in other words, special characters are represented by the words $0100 and higher.

The following table shows the key codes for an implementation using a PC-compatible keyboard:

KEYBOARD KEY NO.	KEY	(1) NORMAL	(2) SHIFT	(3) CONTROL	(4) NUM LOCK
1	ESCAPE	1B/001B	1B/001B	1B/001B	*
2	1 !	31/0031	21/0021	--/0431	*
3	2 @	32/0032	40/0040	00/0100	*
4	3 #	33/0033	23/0023	--/0433	*
5	4 $	34/0034	24/0024	--/0434	*
6	5 %	35/0035	25/0025	--/0435	*
7	6 ^	36/0036	5E/005E	1E/001E	*
8	7 &	37/0037	26/0026	--/0437	*
9	8 *	38/0038	2A/002A	--/0438	*
10	9 (	39/0039	28/0028	--/0439	*
11	0)	30/0030	29/0029	--/0430	*
12	- _	2D/002D	5F/005F	1F/001F	*
13	= +	3D/003D	2B/002B	--/043D	*
14	BACKSPACE	08/0008	08/0008	7F/007F	*
15	TAB	09/0009	--/021F	--/031F	*
16	q Q	71/0071	51/0051	11/0011	*
17	w W	77/0077	57/0057	17/0017	*
18	e E	65/0065	45/0045	05/0005	*
19	r R	72/0072	52/0052	12/0012	*
20	t T	74/0074	54/0054	14/0014	*
21	y Y	79/0079	59/0059	19/0019	*
22	u U	75/0075	55/0055	15/0015	*
23	i I	69/0069	49/0049	09/0009	*
24	o O	6F/006F	4F/004F	0F/000F	*
25	p P	70/0070	50/0050	10/0010	*
26	[{	7B/007B	5B/005B	1B/001B	*
27	] }	7D/007D	5D/005D	1D/001D	*
28	RETURN	0D/000D	0D/000D	0D/000D	*
29	CONTROL	--/----	--/----	--/----	--/----
30	a A	61/0061	41/0041	01/0001	*
31	s S	73/0073	53/0053	13/0013	*
32	d D	64/0064	44/0044	04/0004	*
33	f F	66/0066	46/0046	06/0006	*
34	g G	67/0067	47/0047	07/0007	*
35	h H	68/0068	48/0048	08/0008	*
36	j J	6A/006A	4A/004A	0A/000A	*
37	k K	6B/006B	4B/004B	0B/000B	*
38	l L	6C/006C	4C/004C	0C/000C	*
39	; :	3B/003B	3A/003A	--/043B	*
40	' "	27/0027	22/0022	--/0427	*
41	` ~	60/0060	7E/007E	--/0460	*
42	LEFT SHIFT	--/----	--/----	--/----	--/----
43	\ |	5C/005C	7C/007C	1C/001C	*
44	z Z	7A/007A	5A/005A	1A/001A	*
45	x X	78/0078	58/0058	18/0018	*
46	c C	63/0063	43/0043	03/0003	*
47	v V	76/0076	56/0056	16/0016	*
48	b B	62/0062	42/0042	02/0002	*
49	n N	6E/006E	4E/004E	0E/000E	*

50	m M	6D/006D	4D/004D	0D/000D	*
51	, <	2C/002C	3C/003C	--/042C	*
52	. >	2E/002E	3E/003E	--/042E	*
53	/ ?	2F/002F	3F/003F	--/042F	*
54	RIGHT SHIFT	--/----	--/----	--/----	--/----
55	PRT SCR *	2A/011D	2A/002A	--/031D	2A/002A
56	ALT	--/----	--/----	--/----	--/----
57	SPACE	20/0020	20/0020	20/0020	*
58	CAPS LOCK	--/----	--/----	--/----	--/----
59	F1	--/0101	--/0201	--/0301	*
60	F2	--/0102	--/0202	--/0302	*
61	F3	--/0103	--/0203	--/0303	*
62	F4	--/0104	--/0204	--/0304	*
63	F5	--/0105	--/0205	--/0305	*
64	F6	--/0106	--/0206	--/0306	*
65	F7	--/0107	--/0207	--/0307	*
66	F8	--/0108	--/0208	--/0308	*
67	F9	--/0109	--/0209	--/0309	*
68	F10	--/010A	--/020A	--/030A	*
69	NUM LOCK	--/----	--/----	--/----	--/----
70	SCROLL LOCK	--/011E	--/021E	--/031E	*
71	HOME	--/0120	37/0037	--/0320	37/0037
72	UP ARROW	0B/0125	38/0038	--/0325	38/0038
73	PG UP	--/0123	39/0039	--/0323	39/0039
74	GREY MINUS	2D/002D	2D/002D	--/032D	2D/002D
75	LEFT ARROW	08/0128	34/0034	--/0328	34/0034
76	5	--/0127	35/0035	--/0327	35/0035
77	RIGHT ARROW	09/0126	36/0036	--/0326	36/0036
78	GREY PLUS	2B/002B	2B/002B	--/033B	2B/002B
79	END	--/0122	31/0031	--/0322	31/0031
80	DOWN ARROW	0A/0127	32/0032	--/0327	32/0032
81	PG DN	--/0124	33/0033	--/0324	33/0033
82	INSERT	--/012A	30/0030	--/032A	30/0030
83	DEL	--/012B	2E/002E	--/032B	2E/002E

NOTES:

1. All codes in columns (1) through (4) are hex numbers.
2. The notation AA/BBCC means that the code AA is generated using the normal character input routine (INCH8 in HUMBUG; GETCH, INNOEC, or ICNTRL function 0002 in SK*DOS), and BBCC is generated using ICNTRL (in HUMBUG, or ICNTRL function 0005 in SK*DOS)
3. If BB is 00, then CC is the standard ASCII code for that key, and is equal to AA. With some exceptions, BB codes of 01 stand for unshifted characters, 02 stand for shifted characters, 03 stand for control characters, and 04 stand for characters which do not fit any of the above groups.
4. -- or ---- means that no key code is generated for that key.
5. Items labelled * are not affected by the NUM LOCK key; their key codes are indicated in the other three columns at all times.
6. CAPS LOCK affects only the alpha keys A-Z. For these keys, it reverses the meanings of columns (1) and (2).
7. The ALTernate key adds $80 (or $0080) to all codes shown.
8. The precedence is (a) ALT affects all codes, (b) NUM LOCK codes are not affected by SHIFT or CONTROL, (c) CONTROL is not affected by SHIFT.

9. Codes for unshifted PRT SCRN, and up, down, left, and right arrows were changed from -- to their current values in SK*DOS version 2.4.

OCNTRL Assignments

OCNTRL is the opposite of ICNTRL - it is used to send special arguments to an output device and its driver. Its main purpose is to allow a program to drive a variety of output devices in a common way, without having to be concerned with the particular type of output device being used.

Interfacing to a variety of output devices is again done via a translation table. For example, when a program places the code $0002 into D4 and then calls OCNTRL, the driver uses the translation table to convert the $0001 into whatever character (or sequence of characters) is needed to erase the screen on the current output device.

OCNTRL expects a 16-bit word as an argument and, depending on the exact I/O device, will recognize the following:

$0000	Return number of current driver (0-7) in D5
0001	Return the name of current driver (such as CONS) in D5
0002	Erase screen
0003	Home cursor
0004	Cursor up
0006	Cursor right
0007	Bell (may also use $07 in data stream)
0008	Cursor left (may also use $08 in data stream)
0009	Horizontal tab
000A	Line feed (may also use $0A in data stream) / cursor down
000B	Clear current line
000C	Erase screen and home cursor (form feed on printer)
000D	Carriage return (may also use $0D in data stream)
000E	Mask output of character in D5
000F	Permit output of character in D5
0010	Return size of text screen in (D5, D6)
0011	Move cursor to position (D5, D6)
0012	Return cursor position in (D5, D6)
0013	Return character under cursor in D5
0014	Clear to end of line
0015	Clear to end of screen
0016	Erase screen
0017	Home cursor
0018	Insert line before current line
0019	Delete current line and close up
0020	Switch to normal print / normal intensity
0021	Switch to condensed print
0022	Switch to expanded print
0023	Switch to double strike / bold / double intensity
0024	Switch to enhanced print mode
0025	Switch to italics
0026	Switch to character spacing in D5
0027	Switch to line spacing in D5
0040	Switch to graphics mode in D5

0041 Switch to text mode
0042 Choose color in D5
0043 Draw to position (D5, D6)
FFFx Switch to driver x

Arguments $0000, 0001, and $FFFx (and 0002 for ICNTRL) are handled internally by SK*DOS; all other arguments are passed directly to the ICNTRL or OCNTRL section of the appropriate driver (except for the default console driver in the BIOS, which simply ignores the control codes.)

CUSTOM DEVICE DRIVERS

The SK*DOS system disk includes source code for three device drivers, called SERIAL.TXT, PARALLEL.TXT, and ADM-3A.TXT. These drivers demonstrate the exact format of a device driver, and should be used as examples in case you need to write your own.

SERIAL and PARALLEL are simple drivers for a serial port using a 68681 DUART, and a parallel port using a 68230 PI/T or 6821 PIA, respectively. ADM-3A is an expanded serial driver which shows how ICNTRL and OCNTRL functions are implemented with translation tables.

NOTE: To preserve consistency between different implementations of SK*DOS, users writing device drivers which implement ICNTRL and/or OCNTRL are urged to adhere to the codes listed above. In order to correlate your work with that of others, please contact Star-K Software Systems Corp. before making any extensions to these codes.

KEYBOARD TYPEAHEAD BUFFER

Some SK*DOS implementations (depending on the BIOS or device drivers) may implement a keyboard typeahead buffer; hence there are three DOS calls (STATU1, INNOE1, and FLUSHT) specifically dealing with the operation of such a buffer. If a typeahead buffer is not implemented, then STATU1 and INNOE1 behave exactly the same as STATUS and INNOEC, respectively, while FLUSHT does nothing.

If a typeahead buffer is implemented, then either the BIOS or the device driver, or both, may contain a buffer area which holds incoming characters from the keyboard or other device. This buffer, which is typically 64 bytes but may be larger or smaller, acts as a FIFO or first-in, first-out memory. As characters come in, they go into one end of the buffer; as SK*DOS or a user program needs them, they come out the other end of the buffer. Think of the FIFO as a pipe - characters go in one end and come out the other in the exact same order as they went in. In this way, characters that came in while the system is doing other tasks are held until they are needed. This allows you to type ahead of the computer, for example, giving it commands before they are needed.

When typeahead is implemented, normal input (via GETCH, INNOEC, or INLINE, for example) goes through the buffer. Hence normal programs need not even know whether such a buffer exists or not. Occasionally, however, it is useful to be able to bypass the buffer - when immediate response to an ESCape or control-C is needed, for example - or even to be able to empty the buffer (such as when asking "OK to delete - Y or N?", and you wish to make sure that a previously-entered Y does not get used as an answer.) This is where the three typeahead operations come in. Let us discuss them one by one.

When typeahead is implemented, GETCH and INNOEC input the *first* character in the buffer (i.e., the character that has been in the buffer the longest), whereas INNOE1 inputs the *last* character (the most recent one).

INNOE1 should therefore be used if you are checking for control-C or ESCape, in which case you want to bypass any characters preceding it and act on it immediately. Note that using INNOE1 to get the last character does not remove it from the buffer - it is still there, and still in its correct sequence.

GETCH, INNOEC, and INNOE1 will all wait for a character to be typed if none exists when they are called. Hence we need a way of checking whether there is a character there. STATUS is therefore used to test whether there is any character in the typeahead buffer, whereas STATU1 tests whether a *last character* exists. Note: when there is a single character in the buffer, it becomes both the *first* and *last* character, and STATUS and STATU1 will both return a 'true' when tested. If the character is obtained with GETCH or INNOEC, then both STATUS and STATU1 will go 'false'. But if it is obtained with INNOE1, STATU1 will go 'false' but STATUS will still be 'true', indicating that it is still in the buffer as well.

Finally, FLUSHT is a function which empties the entire buffer so that both STATUS and STATU1 return 'false'. FLUSHT should always be used after INNOE1 returns a break or ESCape character, and may also be used when you want to make sure that some left-over character in the buffer does not provide an undesired answer (as when asking a "Y or N" question.)

The following example shows how these DOS calls are used inside UBASIC to check for a control-C or ESCape. Although checking for ESCape is normally done by SK*DOS internal routines, note that here UBASIC does its own checking. This is needed since it would otherwise remove the ESCape from the buffer before SK*DOS has a chance to check it.

Each time UBASIC outputs a character, or each time it finishes interpreting the current statement, it calls the following BREAK routine:

```
BREAK     DC STATS1              Check 'last' character
          BEQ.S BREAK1           exit if nothing there
          DC INNOE1              else get the last character
BREAK0    CMP.B #3,D5            is it control-C?
          BEQ.S CNTRLC             Yes, go process it
          CMP.B ESCAPC(A6),D5    is it an ESCape?
          BEQ.S ESCAP              Yes, go process it
BREAK1    RTS                    Neither, so exit

CNTRLC    DC FLUSHT              on control-C, flush buffer
          BRA.L READY            and go to READY prompt

ESCAP     DC FLUSHT              On ESCape, flush buffer
ESCAP1    DC INNOEC              Then get the next character
          CMP.B ESCAPC(A6),D5    is it another ESCape?
          BEQ.S BREAK1             Yes, so RTS to continue
          CMP.B #$0D,D5          is it carriage return?
          BEQ.L READY              Yes, so go to READY prompt
          BRA.S ESCAP1           No, so wait for another char
```

Note how the typeahead buffer is flushed only after either the control-C or ESCape is identified; any other character is left in the buffer.

APPENDIX A. USER-ACCESSIBLE VARIABLES

SK*DOS variables of interest to the machine language programmer are listed below. They should be addressed using indexed addressing with A6.

LISTED IN ORDER BY VARIABLE NAME

NAME	ADDRESSES	FUNCTION
BACKSC	736(A6)	Backspace character ($08)
BREAKA	762-765(A6)	Break (Escape) address (long word)
BSECHO	743(A6)	Backspace echo ($08)
CDAY	751(A6)	Current date - day
CMFLAG	793(A6)	Command flag
CMONTH	750(A6)	Current date - month
COMTAB	754-757(A6)	Pointer to command table (long word)
CURRCH	766(A6)	Last character read from buffer
CYEAR	752(A6)	Current date - year
DELETC	737(A6)	Delete character ($18)
DEVERR	3276(A6)	Current error device (1)
DEVIN	3274(A6)	Current input device (0)
DEVOUT	3275(A6)	Current output device (0)
DEVTAB	3278(A6)	I/O device descriptor table
DOSORG	838(A6)	Absolute ORG of SK*DOS
ECHOFL	800(A6)	Input echo flag
ENDLNC	738(A6)	End of line character ($3A)
ENVRON	4074(A6)	1K of environment space
ERRTYP	782(A6)	Error type
ERRVEC	834(A6)	Alternate ERRCODES.SYS vector
ESCAPC	746(A6)	Escape char ($1B)
EXECAD	776-779(A6)	ML execution address (long word)
EXECFL	774(A6)	Execution address flag
FCBPTR	4006(A6)	Pointer to first open FCB (long word)
FIADDR	788-791(A6)	File input address vector (long word)
FNCASE	801(A6)	File Name case flag
FOADDR	784-787(A6)	File output address vector (long word)
LASTRM	753(A6)	Last terminator
LINBUF	608(A6)	Line buffer (128 bytes)
LPOINT	758-761(A6)	Pointer to line buffer (long word)
MAXDRV	802(A6)	Maximum drive number
MEMEND	796-799(A6)	Last usable memory address (long word)
NULLWT	3324(A6)	Null wait constant
OCOLUM	3328(A6)	Current output column
OFFSET	770-773(A6)	Offset load address (long word)
PAUSEB	3326(A6)	Output pause control byte
PLINES	3322(A6)	Number of printed lines per page
PREVCH	767(A6)	Previous character read
PWIDTH	3323(A6)	Page column width
REPEAC	749(A6)	Repeat character ($01)
SEQNO	806(A6)	Sequence number
SLINES	3325(A6)	Number of skipped lines between pages

SPECIO	792(A6)	Special I/O Indicator
SYSDIR	744(A6)	System default directory
SYSTDR	747(A6)	System default drive
USRFCB	0(A6)	User FCB (608 bytes)
USRSPC	4010(A6)	64 bytes of free space for user programs
WORKDR	748(A6)	Working default drive
WRKDIR	745(A6)	Working default directory

LISTED IN ORDER BY ADDRESS

NAME	ADDRESSES	FUNCTION
USRFCB	0(A6)	User FCB (608 bytes)
LINBUF	608(A6)	Line buffer (128 bytes)
BACKSC	736(A6)	Backspace character ($08)
DELETC	737(A6)	Delete character ($18)
ENDLNC	738(A6)	End of line character ($3A)
BSECHO	743(A6)	Backspace echo ($08)
SYSDIR	744(A6)	System default directory
WRKDIR	745(A6)	Working default directory
ESCAPC	746(A6)	Escape char ($1B)
SYSTDR	747(A6)	System default drive
WORKDR	748(A6)	Working default drive
REPEAC	749(A6)	Repeat character ($01)
CMONTH	750(A6)	Current date - month
CDAY	751(A6)	Current date - day
CYEAR	752(A6)	Current date - year
LASTRM	753(A6)	Last terminator
COMTAB	754-757(A6)	Pointer to command table (long word)
LPOINT	758-761(A6)	Pointer to line buffer (long word)
BREAKA	762-765(A6)	Break (Escape) address (long word)
CURRCH	766(A6)	Last character read from buffer
PREVCH	767(A6)	Previous character read
OFFSET	770-773(A6)	Offset load address (long word)
EXECFL	774(A6)	Execution address flag
EXECAD	776-779(A6)	ML execution address (long word)
ERRTYP	782(A6)	Error type
FOADDR	784-787(A6)	File output address vector (long word)
FIADDR	788-791(A6)	File input address vector (long word)
SPECIO	792(A6)	Special I/O Indicator
CMFLAG	793(A6)	Command flag
MEMEND	796-799(A6)	Last usable memory address (long word)
ECHOFL	800(A6)	Input echo flag
FNCASE	801(A6)	File Name case flag
MAXDRV	802(A6)	Maximum drive number
SEQNO	806(A6)	Sequence number
ERRVEC	834(A6)	Alternate ERRCODES.SYS vector
DOSORG	838(A6)	Absolute ORG of SK*DOS
PLINES	3322(A6)	Number of printed lines per page
PWIDTH	3323(A6)	Page column width
NULLWT	3324(A6)	Null wait constant

SLINES	3325(A6)	Number of skipped lines between pages
PAUSEB	3326(A6)	Output pause control byte
OCOLUM	3328(A6)	Current output column
DEVIN	3274(A6)	Current input device (0)
DEVOUT	3275(A6)	Current output device (0)
DEVERR	3276(A6)	Current error device (1)
DEVTAB	3278(A6)	I/O device descriptor table
FCBPTR	4006(A6)	Pointer to first open FCB (long word)
USRSPC	4010(A6)	64 bytes of free space for user programs
ENVRON	4074(A6)	1K of environment space

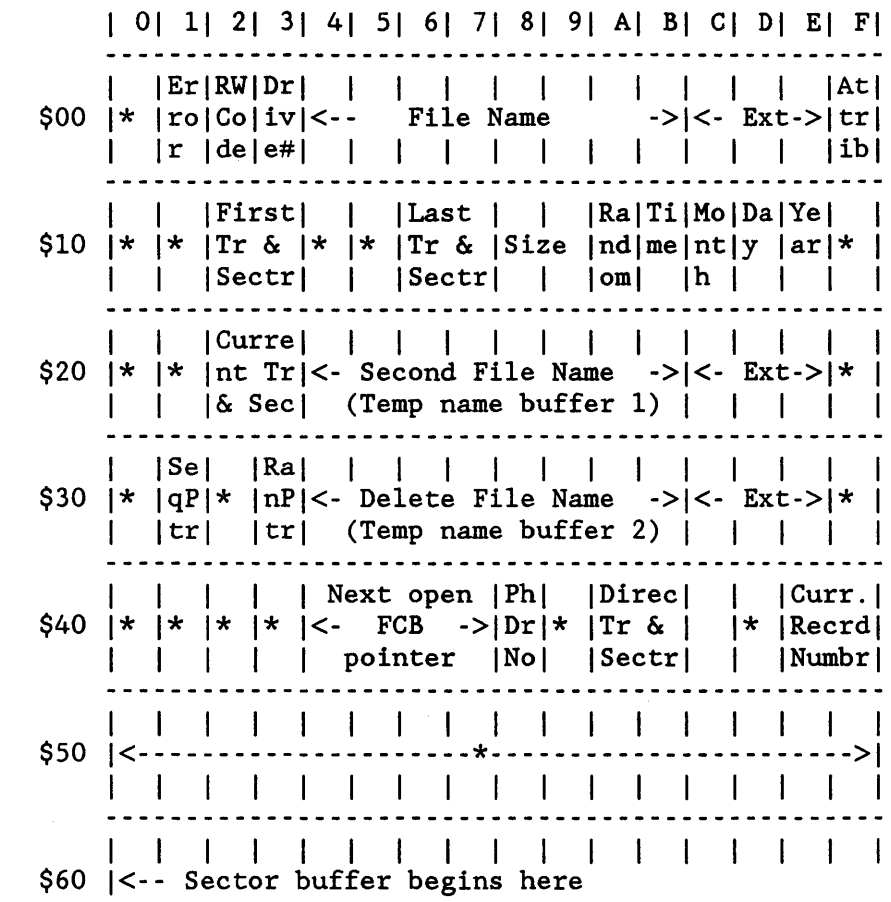
APPENDIX B. THE FILE CONTROL BLOCK (FCB)

The first 96 bytes of an FCB (numbered 0 through 95 for this discussion) hold the following information:

NAME	BYTE(S)	CONTENTS
	0	Reserved for future use
FCBERR	1	Error code (see Appendix E)
FCBRW	2	Read / Write / Update status
FCBDRV	3	Drive number (0 through 9)
FCBNAM	4-11	File name (8 bytes)
FCBEXT	12-14	Extension (3 bytes)
FCBATT	15	File attributes
	16-17	Reserved for future use
FCBFTR	18	First track of file
FCBFSE	19	First sector of file
	20-21	Reserved for future use
FCBLTR	22	Last track of file
FCBLSE	23	Last sector of file
FCBSIZ	24-25	Number of sectors in the file
FCBRAN	26	Random file indicator
FCBTIM	27	Time or sequence number
FCBMON	28	Month of file creation (1 through 12)
FCBDAY	29	Day of file creation (1 through 31)
FCBYR	30	Year of file creation (last two digits)
	31-33	Reserved for future use
FCBCTR	34	Current track number
FCBCSE	35	Current sector number
FCBNMB	36-46	Temporary name buffer 1
	47-48	Reserved for future use
FCBDPT	49	Sequential data pointer to next byte (4 through 255)
	50	Reserved for future use
FCBRIN	51	Random data pointer to next byte (4 through 255)
FCBNMS	52-62	Temporary name buffer 2
FCBCOL	58	Column position (for use by Basic)
FCBSCF	59	Space compression indicator
FCBSPT	60	Number of sectors per track
	61-63	Temporary storage
	64-67	Reserved for future use
FCBLST	68-71	Next FCB pointer
FCBPHY	72	Physical drive number
	73	Reserved for future use
FCBDIT	74	Directory track number
FCBDIS	75	Directory sector number
	76-77	Reserved for future use
FCBCRN	78-79	Current or desired sector number
	80-95	Reserved for future use
FCBDAT	96	Beginning of data area

The names listed in the above table are those used in the SKEQUATE file; it is convenient to use these names rather than numbers when referring to specific FCB bytes in user programs.

The following chart gives a concise summary of this data:



Bytes marked with * are reserved for future use.

APPENDIX C. NON-DISK FUNCTIONS

This appendix lists the non-disk function calls for SK*DOS. They are listed twice, once in order by name and once by op code.

LISTED IN ORDER BY FUNCTION NAME

NAME	OP CODE	FUNCTION
CLASFY	$A02E	Classify alphanumeric characters
DECIN	$A030	Input decimal number
DEFEXT	$A024	Default extension
EXECSD	$A01F	Execute a SK*DOS command
FINDEV	$A012	Find device number from name
FLUSHT	$A044	Flush Type-ahead buffer, if any.
FNPRNT	$A045	Print file directory/name.extension
GETCH	$A029	Get input character with echo
GETDNT	$A03F	Get date and time into
GETNAM	$A023	Get file name into FCB
GETNXT	$A02D	Get next character from buffer
HEXIN	$A02F	Input hexadecimal number
ICNTRL	$A028	Input control (see Chapter 14)
INLINE	$A02C	Input into line buffer
INNOEC	$A02A	Get input character without echo (with TA)
INNOE1	$A043	Get input character without echo (bypass TA)
INTDIS	$A040	Disable interrupts
INTENA	$A041	Re-enable interrupts to previous status
LOADML	$A022	Load open machine language file
OCNTRL	$A032	Output control (see Chapter 14)
OUT10D	$A039	Output 5 decimal digits
OUT2H	$A03A	Output 2 hex digits
OUT4H	$A03B	Output 4 hex digits
OUT5D	$A038	Output 5 decimal digits
OUT8H	$A03C	Output 8 hex digits
PCRLF	$A034	Print CR/LF
PERROR	$A037	Print error code
PNSTRN	$A036	Print string (Without CR/LF)
PSTRNG	$A035	Print CR/LF and string
PUTCH	$A033	Output character
RENTER	$A025	Re-enter SK*DOS
RESIO	$A020	Reset I/O pointers
RESTRP	$A021	Reset trap vectors
STATUS	$A02B	Check keyboard for character (with TA)
STATU1	$A042	Check keyboard for character (bypass TA)
TOUPPR	$A031	Convert lower case to upper (in D5!)
VPOINT	$A000	Point to SK*DOS variable area
WARMST	$A01E	Warm start

LISTED IN ORDER BY OP CODE

NAME	OP CODE	FUNCTION
VPOINT	$A000	Point to SK*DOS variable area
FINDEV	$A012	Find device number from name
WARMST	$A01E	Warm start
EXECSD	$A01F	Execute a SK*DOS command
RESIO	$A020	Reset I/O pointers
RESTRP	$A021	Reset trap vectors
LOADML	$A022	Load open machine language file
GETNAM	$A023	Get file name into FCB
DEFEXT	$A024	Default extension
RENTER	$A025	Re-enter SK*DOS
ICNTRL	$A028	Input control (see Chapter 14)
GETCH	$A029	Get input character with echo
INNOEC	$A02A	Get input character without echo (with TA)
STATUS	$A02B	Check keyboard for character (with TA)
INLINE	$A02C	Input into line buffer
GETNXT	$A02D	Get next character from buffer
CLASFY	$A02E	Classify alphanumeric characters
HEXIN	$A02F	Input hexadecimal number
DECIN	$A030	Input decimal number
TOUPPR	$A031	Convert lower case to upper (in D5!)
OCNTRL	$A032	Output control (see Chapter 14)
PUTCH	$A033	Output character
PCRLF	$A034	Print CR/LF
PSTRNG	$A035	Print CR/LF and string
PNSTRN	$A036	Print string (Without CR/LF)
PERROR	$A037	Print error code
OUT5D	$A038	Output 5 decimal digits
OUT10D	$A039	Output 5 decimal digits
OUT2H	$A03A	Output 2 hex digits
OUT4H	$A03B	Output 4 hex digits
OUT8H	$A03C	Output 8 hex digits
GETDNT	$A03F	Get date and time
INTDIS	$A040	Disable interrupts
INTENA	$A041	Re-enable interrupts to previous status
STATU1	$A042	Check keyboard for character (without TA)
INNOE1	$A043	Get input character without echo (without TA)
FLUSHT	$A044	Flush Type-ahead buffer, if any.
FNPRNT	$A045	Print file directory/name.extension

APPENDIX D. DISK FUNCTIONS

This appendix lists the disk function calls for SK*DOS. They are listed twice, once in order by name and once by op code.

LISTED IN ORDER BY NAME

NAME	OP CODE	FUNCTION
DIROPN	$A00B	Open directory file
DIREST	$A026	Disk restore to track 0
DISEEK	$A027	Disk seek
DSREAD	$A00D	Read directory or system information sector
DSWRIT	$A00E	Write directory or SIS entry
FCLOSE	$A008	Close file
FCSCLS	$A009	Close all open files
FCSINI	$A01B	Initialize File Control System
FDELET	$A00F	Delete a file
FDRIVE	$A01A	Find next drive number
FOPENR	$A005	Open a file for read
FOPENU	$A007	Open a file for update
FOPENW	$A006	Open a file for write
FRBACK	$A015	Backup to previous sector
FREAD	$A001	Read the next byte from file
FRENAM	$A010	Rename a file
FREWIN	$A00A	Rewind file
FRGET	$A016	Read a random byte
FRPUT	$A017	Write a random byte
FRRECD	$A014	Select a specified random sector
FSKIP	$A011	Skip current sector
FWRITE	$A002	Write the next byte to the file
SISOPN	$A00C	Open system information sector
SREAD	$A01C	Read a single sector
SWRITE	$A01D	Write a single sector

LISTED IN ORDER BY OP CODE

NAME	OP CODE	FUNCTION
FREAD	$A001	Read the next byte from file
FWRITE	$A002	Write the next byte to the file
FOPENR	$A005	Open a file for read
FOPENW	$A006	Open a file for write
FOPENU	$A007	Open a file for update
FCLOSE	$A008	Close file
FCSCLS	$A009	Close all open files
FREWIN	$A00A	Rewind file
DIROPN	$A00B	Open directory file
SISOPN	$A00C	Open system information sector
DSREAD	$A00D	Read directory or system information sector
DSWRIT	$A00E	Write directory or SIS entry
FDELET	$A00F	Delete a file
FRENAM	$A010	Rename a file
FSKIP	$A011	Skip current sector
FRRECD	$A014	Select a specified random sector
FRBACK	$A015	Backup to previous sector
FRGET	$A016	Read a random byte
FRPUT	$A017	Write a random byte
FDRIVE	$A01A	Find next drive number
FCSINI	$A01B	Initialize File Control System
SREAD	$A01C	Read a single sector
SWRITE	$A01D	Write a single sector
DIREST	$A026	Disk restore to track 0
DISEEK	$A027	Disk seek

APPENDIX E. SK*DOS ERROR CODES

SK*DOS uses the following error codes; in addition, user programs may use other error codes which are documented in their respective manuals.

NUMERIC CODE	MEANING
1	FCS operation code error
2	File already open or in use
3	File already exists
4	File does not exist
5	Directory Error
7	Disk is full
8	Input past end of file
9	Disk read error
10	Disk write error
11	Disk is write protected
12	Protected file
13	Error in closing file
14	Disk seek error
15	Invalid drive number
16	Drive not ready
18	This FCS operation not permitted on this file
19	Random file operation not allowed
20	Disk I/O error
21	Illegal or missing file name or extension
22	Can't Close error
23	Random file map overflow
24	Specified random sector number is invalid
25	Random sector number does not match file contents
26	SK*DOS command syntax error
28	Missing transfer address
29	Disk has been switched while file was open
30	File not open

Internally, SK*DOS errors are represented by one-byte numbers which are generally placed into byte 1 (the second byte) of a file control block by the File Control System. User programs should test for these by their numbers.

Error messages are generally printed out by using the PERROR routine, which prints out the error number, usually followed by a one-line explanation of the error.

Error 1 (FCS operation code error) is treated a bit differently from the others. When it is encountered, SK*DOS immediately prints an error message and asks whether you wish to continue anyway. If you answer Y, then it will return error 1 to the calling program and continue; any other answer will immediately abort the program, close all files, and return to SK*DOS.

The one-line explanations of errors are obtained from the ERRCODES.SYS file, and will only be obtained if this file is on the system disk; otherwise only the error number will appear. (The ERRCODES.SYS file also contains explanations for 68xxx exceptions. If the processor hits an exception, such as a buss error, it will print error 1xx - where xx is the CPU exception vector number - followed by an explanation taken from the ERRCODES.SYS file.)

APPENDIX F. DEFAULT EXTENSION CODES

The following default extension codes are used by the DEFEXT routine.

```
 0 = .BIN   Binary program file
 1 = .TXT   Text file
 2 = .COM   Command file
 3 = .BAS   Basic language file
 4 = .SYS   SK*DOS system file
 5 = .BAK   Backup file
 6 = .SCR   Scratch (temporary) file
 7 = .DAT   Data file
 8 = .BAC   Basic compiled file
 9 = .DIR   Directory
10 = .PRT   Printer file
11 = .OUT   Output file
12 = .BAT   Batch file
13 = .SRC   Assembler source file
14 = .PIP   Pipe
```

NOTE: Note that the default extension for commands is .COM, which is different from the .CMD used in 6809 SK*DOS. This allows 6809 and 68K commands to coexist on the same disk without the danger of accidentally calling the wrong one.

APPENDIX G. SK*DOS COMMAND SUMMARY

This Appendix describes the commands currently supplied with SK*DOS. From time to time, however, we may add other commands which may not be described here. Most commands will provide you with information on their proper usage if you type the command name, a space, and a question mark, as in

```
BUILD ?
```

This Appendix describes the following commands:

Disk-resident commands

ACAT	FIND	PROMPT
APPEND	FORMAT	PROTECT
BACKUP	FROMSDOS	RAMDISK
BEEP	FTOH	REDOFREE
BUILD	HDFORMAT	RENAME
CACHE	HELP	SCAT
CAT	HTOF	SEQUENCE
CHECKSUM	LINK	SK*DOS09
COMPARE	LIST	STEPRATE
COPY	LOCATE	SYSTEM
DAMON	MAKEMPTY	TCAT
DELETE	NOBEEP	TOMSDOS
DEVICE	PARK	TRACENAB
DIFF	PDELETE	UBASIC
DISKNAME	PEEK	UNDELETE
DOSPARAM	POKE	VERIFY
DRIVE		VERSION
EDLIN		WORK

Memory-resident commands

DIR	MON	TRACE***
GET	RESET	XEQ
GETX	SAVE	

Other supplied programs

PICTEST	S1TOCOM

ACAT

The ACAT command displays a fully alphabetized catalog listing of a disk or a directory. It is similar to CAT, except that it does not support all of the options that CAT does.

To use ACAT, type the word ACAT, followed by an optional drive number and directory specifier (which defaults to the work drive if not entered). You may follow this with a *match-list* - additional information if you want only certain files listed. The general format is to enter

```
ACAT <where> <what>
```

where the <where> can be either a drive number such as 0 or 1, or a drive number and directory letter such as 1.C/; a * may be used in place of a directory letter to scan all directories on the disk.

The <what> in the above example is the match-list specifying which files are to be listed. The following examples show some possibilities:

`ACAT` catalogs all files on the current work drive and its default directory

`ACAT 1` catalogs all files on drive 1

`ACAT 1 TX` catalogs all files on drive 1 whose file names begins with the letters TX

`ACAT .COM` catalogs all files with .COM extensions

`ACAT TX.COM` catalogs all files whose file name begins with TX AND which also have .COM extensions

`ACAT TX*E.COM` catalogs all files whose file name begins with TX*E, (where * stands for any character) AND which also have .COM extensions

`ACAT TX .COM` catalogs all files whose name begins TX, and also all files which have .COM extensions.

`ACAT 2.C/ TX .COM` catalogs all files whose name begins TX, and also all files which have .COM extensions, but only on drive 2 directory C.

`ACAT 2.*/ TX` catalogs all files whose name begins TX in all directories of drive 2.

Note how a * is a wild-card character which stands for any character in the middle of a file-spec, and stands for any directory when used in a drive.directory specifier.

Note also that when a drive number is not specified, ACAT defaults to the default work drive and its default subdirectory; if a drive number is specified without a directory code (even if the drive number happens to be the same as the current work drive) then ACAT will search the root directory.

APPEND

The APPEND command is used to combine several 'source' files together to make a single large 'destination' file. For example, it can combine a number of text files together into a large file, or can combine several machine language programs into one large program. APPEND writes a new destination file, with the original source files left unchanged.

To use this command, type the word APPEND followed by the names of the source files to be combined, followed by the name of the resulting destination file. For example, the command

```
SK*DOS: APPEND PROG1.BIN PROG2.BIN 2.PROG3.BIN 1.PROG.COM
```

would combine PROG1.BIN, PROG2.BIN, and 2.PROG3.BIN, in that order, into a new file called PROG.COM on drive 1. (All of the source files must exist, and the destination file must not exist.)

Although files of any type may be appended, usually all the files will be of the same type. The extension of the first source file defaults to .TXT if not specified otherwise, and the extensions of all succeeding files (source and destination) default to the same extension as the first file.

When machine language program files having transfer addresses are appended, the transfer addresses are carried forward into the destination file, but the SK*DOS load routine uses only the last transfer address given.

BACKUP

The BACKUP command is used to make an exact backup of a disk. This command requires two drives.

If BACKUP encounters an error on either the source disk or the destination disk, it will display an error message but continue copying until it finishes the disk. To call BACKUP, enter the command BACKUP followed by the drive numbers of the source and destination drives, as in

```
SK*DOS: BACKUP 0 1
```

This command would copy from drive 0 to drive 1. (The two drive numbers must be entered and must be different.) Note carefully - BACKUP copies from the first drive specified to the second drive.

Before BACKUP can be used, you must format the destination disk with the FORMAT command. Furthermore, the destination disk must have at least as many tracks and sectors as the source disk. If it has fewer tracks or sectors, then BACKUP will display an error message and stop.

After BACKUP is finished, the destination disk will have the same apparent number of tracks and sectors as the source disk. For example: suppose you BACKUP a 35-track single density 5-1/4" disk (ten sectors per track) onto a 77-track double density double sided 8" disk. The destination disk will have only 35 tracks and ten sectors per track. (In reality, the remaining tracks and sectors will still be there, but will be completely inaccessible to SK*DOS from then on.)

BEEP
NOBEEP

Assuming that your terminal supports the BELL character (ASCII $07), the BEEP command will sound the bell (or beep) at each SK*DOS: prompt from then on. This is a useful function if you like to walk away from your computer while it is doing a lengthy task.

Once the BEEP command is given, the bell will sound until the computer is rebooted, or until the NOBEEP command is used to cancel BEEP.

BUILD

BUILD is used to generate a text file on the disk. BUILD is not intended to replace a more general purpose editor; instead, BUILD might be used for testing or generating simple files.

The BUILD command line must include the name of the file to be generated. This is usually done by including a file specification after the word BUILD, as in this example:

```
BUILD TEXT
```

The file specification defaults to a .TXT extension unless specified otherwise, and also assumes the current working drive.

While entering text with the BUILD command, you may correct any line by backspacing and retyping a character. Or, while still in the middle of a line, you may erase the entire line and start it over by hitting the control-X key. Once the line is entered by hitting the carriage return key, however, it is stored and cannot be changed. In other words, BUILD is not an editor.

The BUILD program ignores control characters, and is limited to a maximum line length of 127 characters.

To end entering text, type a # character at the beginning of a new line.

CACHE

The CACHE command is used to set up a disk cache; that is, a memory area which stores data read from or written to a floppy disk. When SK*DOS tries to subsequently read that data again, it reads it from the cache memory rather than reading it from the physical disk itself. This significantly speeds up disk operations.

There are three ways of calling the CACHE command:

SK*DOS: CACHE NEW <memory size in K>

sets up a new cache memory of the specified size. For example, the command CACHE NEW 320K would set up a 320K cache memory area. The specified cache memory can range from 32K to 1024K (actually 1 megabyte) in size.

Another way of calling CACHE is with

SK*DOS: CACHE <drive number>

which erases ('flushes') all data from the cache which corresponds to the specified logical drive number.

Finally, the command

SK*DOS: CACHE STATUS

displays a status report of the cache memory, including the memory size, the number of sectors it can store, the number of sectors currently empty, and the actual number of sectors currently stored for each floppy drive.

IMPORTANT WARNING: Since CACHE has no way of knowing when you insert a new disk into a drive, it is essential that you manually flush the cache each time you swap disks. IF YOU DO NOT DO SO, YOU WILL PROBABLY CORRUPT THE NEW DISK. This will happen because CACHE will think you are still using the old disk, and will read and write data as if the old disk was still in the drive. Any data written to the new disk will go in the locations it would have gone on the old disk.

Note that CACHE only saves data for floppy disks; it does not store data for the RAM disk (since that would just duplicate data already in RAM) or a hard disk (since hard disks are generally almost as fast as the cache memory). Hence the memory assigned to CACHE is used only for floppy disk data.

The CACHE program can be used in addition to RAMDISK, but if both are used together, then the RAMdisk must be set up first. The reason for this requirement is that this allows the RAMdisk memory to be above the cache memory. If SK*DOS is subsequently rebooted, there is a greater chance of the RAMdisk data being preserved since it is in higher memory. The cache memory, on the other hand, is always erased when rebooting. (This also serves to explain why the cache memory size can be changed while RAMdisk memory size cannot.)

Although CACHE can use anywhere from 32K to 1024K (i.e., one megabyte) of memory, the actual size chosen depends on the application. If you merely intend to use a floppy disk for a few files, then 32K or 64K might be sufficient. If, on the other hand, you intend to do extensive processing with one disk, then the size should be about 20% larger than the size of the floppy disk (since CACHE needs some extra memory for its data storage and for 'elbow room'.) If you have several floppy drives, then the cache memory should be large enough to accommodate all of the expected floppy disk operations on all drives.

When you intend to use a single floppy for extensive operations, you can greatly speed up processing by reading the entire floppy into the cache at the beginning (assuming that the cache memory is large enough to hold the disk's contents.) This is easily done by doing a CHECKSUM on the disk, which reads the entire disk and (coincidentally) stores it in the cache memory. Once this is done, SK*DOS will no longer need to read that disk at all; it will only access the disk for writing.

Note that CACHE only stores floppy data; in fact, it goes by the physical drive number rather than the logical drive number. Hence if you use the DRIVE command to reassign a drive's logical drive number (but don't swap diskettes in the drive) CACHE will 'follow' the diskette to its new logical drive number.

FINALLY ... ONCE AGAIN ... MAKE SURE TO FLUSH THE CACHE WHENEVER YOU CHANGE A FLOPPY DISK!!

CAT

CAT is used to display the contents ('catalog') of a disk or directory.

At its simplest, invoking the CAT command can be as simple as typing the word CAT , or it can be followed by a number of parameters. A more complex command might be in the form

```
CAT  <how> <where> <what>
```

All of these parameters are optional; any of them can be used, but if you use more than one then they must be in the order shown above: *how, where, what.* All parameters are entered on the command line. If you have trouble remembering what to do, the CAT help list can be displayed by entering a command of CAT ?.

<How> Options

The so-called "how" options consist of one or more option letters preceded by a + sign, and should follow after the word CAT, as in

```
CAT +SDF1
```

The available options are:

A	Alpha	Alphabetize output by 1st letter
D	Date	Display file Date
F	File No.	Display File number
M	Maximum	Full listing with all options
N	Non-existent	Display Deleted files with (-)
P	Paging	Paging with printer column width
R	Repeat	Repeat CAT as listed on command line
S	Size	Display file Size in sectors

CAT normally defaults to the M or Maximum option when none is specified. Entering any option letter on the command line will turn that option on, but all others are turned OFF. Hence you must enter ALL options you want if you enter any at all.

The following gives more information on specific options:

A - This option will group the listing by the first letter, rather than provide a fully alphabetized list (such as provided by ACAT). When the "A" option is used you cannot use any match strings.

D - This option will display the file's creation date. If the month is zero or over 12, CAT will display the month as "BAD".

F - This option will list the actual directory number.

M - The "M" option will provide information on track-sector data and the protect file code information. The "M" option turns on the D,F and S options plus routines needed for the header, track and sector, and protect codes.

N - This option will display deleted file entries that exist in the directory. The actual file may not exist on the disk - it may have been over-written when it was part of the list of free sectors. You cannot assume a deleted file is intact unless you know it was recently deleted and you have not written enough new files on the disk to reuse the deleted file's sectors. CAT will display a dash (-) as the first character of the file name. This option is useful if you intend to try to rescue a deleted file with the COPY command.

P - This option will enable a paging subroutine to place blank lines on the top and bottom of the page. This is handy when listing a disk with a large number of files. The printer width equate is picked up and used to calculate the number of columns allowed for each form of the printed listing.

R - The "R" option allows CAT to re-start. A prompt will appear at the end of each CAT; enter "E" to exit to SK*DOS, or hit any other key to do another CAT as originally listed on the command line. This option will allow you to change disks and do a CAT without re-entering the data on the command line. The user can select CAT options to control the type of information listed in the disk file.

S - The "S" option will enable the sector size of files to be displayed.

Since the M option is the normal default, the typical CAT display will look like the following:

```
Drive: 2 Dir: A/  Disk: SOURCE 4 Created: 31-May-82

File#   Name     Type     Begin    End     Size  Date       Prt
 2      COPY99   .TXT     29-03    11-0F    128  21-Jun-82  D
 12     RANDOM   .SYS R   01-01    01-03      3   3-Nov-81

Files=21  Biggest=206  Total=131/1029  Free=111
```

The top line of this display provides the drive number, directory, disk name, disk number, and disk creation date.

A separate line describes each file, and provides its number within the disk directory sectors, the name and extension, beginning and ending tracks and sectors, size, and creation date. Random files are identified by the letter R to the right of the extension. The three SK*DOS protect codes are indicated by letters (although the catalog protect bit is not honored by CAT): C = catalog protect; D = delete protect; W = write protect.

The last line indicates the total number of files on the disk, the size of the biggest file on the disk, the total size of the displayed files as compared with the total size of all files on that disk, and the number of free sectors.

<Where> Parameter

The "where" parameter specifies which disk or directory to display. This parameter can be either a plain drive number (such as 1) or a drive number, period, directory letter, and a slash (as in 1.C/). An asterisk (*) can be used to mean "all directories" (as in 1.*/).

It is not possible to enter a directory without a drive number preceding it. If no "where" information is entered, then the default work drive and directory are chosen; if only a drive number is entered, then the root directory of that drive is chosen.

<What> Parameters

The "what" parameter specifies a *match-list* which is compared against the files in the directory, and only those files matching the match-list are displayed. The match-list can contain one or more file-names and/or extensions, or portions of names and extensions. The asterisk (*) is a wild-card character in a match-list which will match against any character in a file name.

Here is an example of an elaborate CAT command:

```
CAT +AM 1.C/ AD .TXT J*M.BAK
```

The A and M options specify an alphabetized maximum listing of files in the C directory of drive 1. Only files satisfying one of the following three criteria will be shown: (1) any file whose name begins with AD, (2) any file having a .TXT extension, or (3) any file whose name begins with J, whose third letter is an M, and which have a .BAK extension.

(The original CAT program and documentation were written by, and provided through the courtesy of, Bruno D. Puglia and Leo E. Taylor.)

CHECKSUM

CHECKSUM is used to generate and display a sum of all the bytes on a disk. The check sum is shown as an 8-digit hex number.

CHECKSUM is used simply to verify that the content of a disk has not changed over some period of time. Although it can be used with floppy or hard disks, we find it most useful to use with the RAMDISK program. Whenever we walk away from the computer for an extended length of time while the RAM disk is in use, we run CHECKSUM on leaving, and then again when we return. This makes sure that the contents of the RAM have not accidentally changed, perhaps due to a momentary power interruption while we were gone.

To use CHECKSUM, simply follow it with the drive number, as in

```
SK*DOS: CHECKSUM 1
```

COMPARE

The COMPARE command requires two drives, and does an exact, byte by byte, comparison of two disks. It is thus useful for checking whether a disk has been backed up correctly (although the disk will be verified during BACKUP.)

To use COMPARE, type the word COMPARE followed by the drive numbers of the two drives holding your disks, as in

```
SK*DOS: COMPARE 0 1
```

which would compare the disks in drives 0 and 1.

If you want to check whether a single disk is readable, you may also specify the same drive number for both disks, as in

```
SK*DOS: COMPARE 0 0
```

This mode reads a single disk twice, and checks not only that it is readable, but also that the same data is read both times.

COPY

COPY is used to copy files from disk to disk, or from directory to directory. It can copy just one file, a group of files, an entire directory, or an entire disk. It can also retrieve deleted files (although UNDELETE is generally easier to use), alphabetize while copying, and more.

In its simplest form, the COPY command is used by entering the word COPY, the file-spec of the file to be copied, and the file-spec of the destination. For example,

```
COPY 0.PROG.TXT 1.NEWPROG.ABC
```

would copy PROG.TXT from drive 0 to a new file called NEWPROG.ABC on drive 1.

But there are many other possible ways of calling COPY. Although there are some exceptions, the most common form of a COPY command looks like this:

```
COPY <how> <from where> <to where> <since date> <match-list>
```

We will leave the *how* and *since date* for later, and begin with the *from where*.

The <from where> parameter

This item describes where the file(s) will come from. It may be

(a) a plain drive number, such as 0 or 2, in which case the current default directory is assumed (except when the F option is used - see later),

(b) a drive number and directory, such as 1.A/ (or 2.*/ where the asterisk would mean all directories), or

(c) a file name and extension, possibly preceded by a drive number and directory. If such a specific file-spec is supplied, then a match-list would not be used. An extension is always required in this case, as this option must narrow down the copy function to a specific file. If a drive and directory are not supplied, then the current default will be used.

The <to where> parameter

This item describes where the file(s) will be copied to. It may be any one of the three formats described under <from where> above. If the <to where> information is incomplete, then the drive and directory will come from the default work values, and the file name will be taken from the "from" list. For example, if the command is

```
COPY 0.A/FILE2.TXT 2
```

then the file will be copied to the default directory on drive 2 and will still be called FILE2.TXT. Any other directory would have to be specifically called out. If the "to" directory is specified as */, then the file will be copied to the same directory as it was on the source disk.

The <match-list> parameter

In those cases where the "from" parameter does not provide enough data to specify a specific file, the match list can be used to narrow down the field to a specific file or group of files. This list is essentially a list of one or more file names and/or extensions, or portions thereof. COPY will copy only those files which match the match-list.

If, for example, the match-list consists of the word UN, then any file whose name begins with UN will be copied; if it consists of .TXT then any file with a .TXT extension will be copied. If the match-list item is UN.TXT, then files must both have a name beginning with UN and also must have a .TXT extension. An asterisk can be used as a wild-card character inside a name or extension, as in U*D, which would match with any name such as UAD or UBD or even U8D.

As mentioned earlier, the match-list is optional; if absent, then any file satisfying the <from where> parameter will be copied.

For example, `COPY 0 1 PROG.TXT` would copy PROG.TXT (and all other files whose name begins with the letters PROG and which have .TXT extensions) from drive 0 to drive 1.

`COPY 0 1.D/ A B .BIN` would copy all files whose names begin with A or B, or which have a .BIN extension, from drive 0 to directory 1.D/.

The <how> parameter

The <how> parameter describes how the copy is to be done, and allows a variety of variations on the basic COPY command. It is specified by including one or more option letters between the word COPY and the <from where> parameter, which must always begin with a drive number (so COPY can tell where the option letters end and the <from where> parameter begins. The following option letters are allowed:

A	copy in Alphabetical order
C	allow Corrupt files to be copied
D	copy files with newer Date
E	delete Existing destination file
F	copy by File number (alpha not allowed)
K	Kill duplicate file on source
L	List files without copying
M	Make random file
N	copy files Not on destination
O	turn Off defaults
P	Prompt before copying file
R	Recover from track-sector
S	copy Since a specified date
T	Track zero protection override
U	Use current SK*DOS date
W	Wait for disk change
Z	Zap source file after copying

Here are short explanations of these options:

The A option will alphabetize the source directory before files are selected to be copied.

The C option will enable you to copy a file that is damaged by a CRC error or record sequence error. This is a slightly dangerous option which should only be used if you don't have an alternate copy of a file.

The D option will find files that are on both disks and compare their creation dates and times or sequence numbers. If the source file is newer it will be copied as a replacement for the older destination file.

The E option is used when you want to replace a file on the destination disk. This option will suppress the prompt for whether you wish to delete the existing file. It will often be used along with D to update a disk with newer versions of programs.

The F option changes copy's parameters from a match string list to a list of file numbers. Follow the drive numbers with a list of file numbers for those files that you want to copy. File numbers can be found with CAT. A group of files can be specified as a starting and ending number separated by a dash. The command `COPY F 0,1 5 13-18 9` will copy file 5, files 13 through 18, and file 9, all to the root directory of drive 1. The "from" drive number always refers to *all* directories, but you must append `.*/` if you wish the files copied to the corresponding directories of the "to" drive. This option is useful for copying the contents of a large disk to two or more smaller ones.

The K option is VERY dangerous. This command isn't really a copy; rather it uses the directory compare routines to delete files from the source disk that appear on the destination. This allows you to clear off extra copies of programs not needed on the source disk. It operates very fast and will clear off a number of files faster than you can hit reset. As with all dangerous options it is protected with an `ARE YOU SURE` prompt. `COPY K 1,0` is most effective in killing files on drive 1 when they exist on drive 0. `COPY KD 1,0` will kill the file on drive 1 when it is older than the file on drive 0. Use `COPY KDL 1,0` to preview what files will be deleted.

The L option disables the file copy subroutine. This is used to display a list of files that *would* have been copied if you hadn't used option L. This can be used with other options to check disks for duplicate files, newer dates, bad files, etc.

The M option is used to convert a SK*DOS sequential file into a random file. This option is also used with R to recover a random file by track and sector. NOTE: this option is not used for normal file copying; if the source file is random COPY will automatically make the destination file random.

The N option is used to copy the files on the source disk that are not already on the destination disk. This can be used to add all new files to a backup disk.

The O option is a dummy character used to turn off all default options if you do not want any options. If used with any other option letters it has no effect.

The P option enables this prompt: `Prompt off (P); SK*DOS (S); copy (Y/N)?`. You should respond with P if you want to continue copying without the prompt or S to return to SK*DOS or Y to copy this file. N or any other character will skip to the next file. This is useful for scanning through a disk copying only certain files. Another use is skipping down to a certain file on a disk and copying all files after that.

The R option is used to read a file without using the directory. If the directory of a disk has been destroyed but the user knows the file's starting address, the file can be recovered. The command `COPY R 1 2B 5 0.NEWFILE` will read from drive 1, track $2B, sector $5 until encountering an end of file or a record out of sequence. The write file extension will default to .SCR. A second use for this option is to recover a deleted file (the first sector of a deleted file can be found with the N option of CAT.) If the file has not been over-written, COPY can recover it. Record sequence checking eliminates UNDELETE's restriction that the file be the last file deleted. Provision was made to start copying in the middle of a file. This is a somewhat dangerous option since it allows the user to override the SK*DOS File Control System. The command must be typed as shown with three numbers and a file name. Only a few other options can be used with R. See M if the original file was random.

The S option is used to copy only files generated on or after the date specified in the "since date" parameter. For example, the command `COPY S 0 1 6-28-88` would copy all those files dated June 28th, 1988 or later.

Since no directory is specified, only files in the root directory would be copied in this case. The S option can, however, also be combined with several other COPY command features to make more complex commands. The S option *can not* be combined with the F (copy by file number) option, since the F option will take precedence. (The F and S options are often used for backing up a hard disk to floppy disks. F is used for a complete backup to copy groups of files at a time to separate floppy disks; S is useful in making *incremental backups* of just those files which have been changed or newly generated in the last few days.)

The T option is used only for those systems that store data files on track zero. COPY normally prevents a file from linking to track zero. Some hard disk systems include track zero as part of their free sector chain; in that case, COPY will switch to the T option automatically.

The U option is used when you want the destination file to have the present SK*DOS date rather than the date of the source file being copied. This may be useful if you know the source file has an erroneous date.

The W option loads COPY, but then waits for any key to be pressed before continuing. This allows you to switch disks before actually copying. (Note that the same effect can be achieved by using GET to load COPY, switching disks, and then using XEQ (plus appropriate arguments) to execute it.)

The Z option is somewhat dangerous. It is used to delete the file from the source disk after it is copied. Essentially the file is moved from one disk to the other.

Option letters are often used in combinations; here are some examples of popular combinations:

DN - Updates the destination with all files from the source that are not on the destination or have an older date on the destination.

EZ - Moves file P.COM from source to destination no matter what.

KD - Kill the file on drive 1 when it is older than the file on drive 0.

LNA - Alphabetically lists those files on the source that are not on the destination.

The <Since date> parameter

This parameter is used only with the S option, and specifies the date used for copying; only files dated on or after this date will actually be copied. The month, day, and year must appear in that order, and may be separated by hyphens or slashes, as in 6-28-88 or 6/28/88. Note that the date must appear *after* the "to where" parameter, but before the "match-list", if used.

General Comments

It is important to realize that if you get an error while writing a file to the destination disk the new file may be defective. The file may appear in the directory but usually it is incomplete.

The most common error message is DATE BAD. This occurs when the user does not enter a valid date when SK*DOS is booted or by failure of a hardware clock when used for setting the SK*DOS date. COPY will check the date on all files when it reads the disk directory and report any dates outside a reasonable range. This reduces the chance that a bad date will be passed on to the new file. There are two alternatives when the BAD DATE message appears. You can answer Y indicating that you approve of bad dates or answer N and not copy the file. After returning to SK*DOS you can re-enter COPY using option U which will assign the current SK*DOS date to the file or use the DATE command to set the file date to the day the file was made.

In order to check that the destination disk exists and is not write protected, COPY reads the SIS (track 0 sector 3) of the destination disk and then duplicates it on sector 4 (which is an unused sector). If the disk is protected or not ready the program will exit immediately.

(The original COPY program and documentation were written by, and provided through the courtesy of, Bruno D. Puglia and Leo E. Taylor.)

DAMON

DAMON is a trouble-shooting program which displays the drive, track, and sector number of each sector being written or read by SK*DOS, and the address of the current File Control Block. You might use it whenever you want to follow SK*DOS disk accesses to see what it is doing.

DAMON substantially slows down the operation of the disk system, since each sector read or written is accompanied by a display of its number on the screen. Moreover, once activated, the only way to turn it off is to reboot the system. Hence it is a specialty program, to be used only when needed to debug a program.

DELETE

DELETE is used to delete a disk file from a disk.

To delete a file, enter the command DELETE followed by the file name, as in

```
DELETE TEXT.TXT
```

The extension is required. The command defaults to the default directory on the work drive.

You may delete several files with one DELETE command by specifying more than one file name. The maximum length of the entire DELETE command, however, is limited to 128 characters.

DEVICE

DEVICE is used to display or install device drivers. To display the current device assignments, simply type the command

```
SK*DOS:  DEVICE
```

and you will get a display similar to the following:

```
The current I/O device assignments are:
 Normal      Device  Device   Driver
  use        number   name
 ------      ------  ------  ---------
Terminal        0     CONS   Default driver
Error device    1     CONS   Default driver
Printer         2     CONS   Default driver
                3     CONS   Default driver
                4     CONS   Default driver
                5     CONS   Default driver
                6     CONS   Default driver
Null device     7     NULL   Default driver
```

To install a new driver, use the format

```
SK*DOS:  DEVICE <driver name> AS <device name> AT <device number>
```

where
...the <driver name> is the name of a device driver file (with a default extension of .DSK) or, alternatively, the word DEFAULT for the default console driver,
...the <device name> is a four-character name such as CONS or PRTR which will be assigned to the device
...the <device number> is a number from 0 to 7 that will be assigned to the driver.

Additional information on DEVICE usage may be found in Chapter 14.

DIFF

DIFF is used to compare the contents of two text files and print out any differences between them.

The correct syntax for using DIFF is

```
SK*DOS: DIFF <file-name-1> <file-name-2>
```

where the file-names are of the two files being compared.

DIFF compares the two files on a line-by-line basis, so two files having the same text but formatted into different lines will be shown as different. Each time DIFF finds lines which are different, it displays them, and continues until it finds two lines which again match.

DIFF has several characteristics which should be noted. If the two lines are of different lengths, then DIFF will stop at the end of the shorter file and print out the message "No further similarities found". So as to avoid resynchronizing on blank lines, it ignores them. When looking forward for another pair of matching lines, it stops after 50 lines. Hence if there are more than 50 consecutive different lines in the two files, DIFF may not be able to resynchronize at the very end of the dissimilar texts.

DIR

DIR is used to display the contents (directory or catalog) of an entire disk, including all its subdirectories.

DIR is not as useful, and does not provide as much information as some of the other 'catalog' utilities such as ACAT, CAT, SCAT, or TCAT. It is, however, memory-resident, so it can be used without having a system disk with another 'cat' utility installed in a drive. Except for an optional drive number, DIR accepts no other options or match list arguments.

In addition to providing a list of files on a disk, DIR provides one additional piece of information which the other programs do not provide - a list of subdirectories used on the disk. This is done in the form of a display like this:

```
Subdirectories used: < ABCDEFGHIJKLMNOPQRSTUVWXYZ >
```

Only those letters which are used as subdirectories appear. The symbols < and > appear only in those cases where subdirectories exist with ASCII codes either below A or above Z. Such names are normally illegal, but may still exist if written by programs.

DISKNAME

The DISKNAME command is used to change the name, number, or date of a disk. Simply type the command DISKNAME, followed by the logical drive number, as in

```
SK*DOS:  DISKNAME 2
```

DISKNAME will then print out the current values, and allow you to enter new values. You may leave the old name or number unchanged by pressing ENTER, or may leave the date unchanged by answering N to the question.

DOSPARAM

The DOSPARAM command can be used to display or change a number of DOS parameters.

To display the current DOS parameters, use the command

```
SK*DOS:  DOSPARAM [<device number>]
```

where the <device number> is optional and will default to 0 if not given. DOSPARAM will then print out several system-wide parameters, as well as parameters for the specified device number, as shown in this example:

```
   SK*DOS PARAMETERS (COMMON TO ENTIRE SYSTEM):
BS Backspace    = $08 = Control-H     BE BS echo      = $08 = Control-H
DL Delete       = $18 = Control-X     EL Endline      = $3A = :
TB Tab          = $00 (none)          ES Escape       = $1B = Control-[
RP Repeat       = $01 = Control-A     MD Max drive no= 1
OF OFFSET       = $00005B00           ME MEMEND       = $000BFFFF
   SK*DOS PARAMETERS FOR DEVICE NUMBER 0
PL Print lines = 0                    SL Skip lines   = 0
WD Width        = 0                   NL Null wait    = 0
PS Pause        = NO                  EF End of File  = $1A = Control-Z
BR Baud rate    = 0                   XF X-OFF Char   = $00 (none)
XN X-ON Char    = $00 (none)
```

To use DOSPARAM to change one or more parameters, type DOSPARAM, followed by an optional device number, followed by the two-letter abbreviation for the parameter to be changed (see below), followed by an equals sign, and then followed by the new value.

Several parameters can be changed in one command line, as in

```
SK*DOS:  DOSPARAM 2 WD=64 BS=$08 PS=N
```

The PAUSE value is either YES, NO, ON, or OFF; all other values are decimal except those identified with a $.

See the chapter on 'User-Accessible Variables' for the explanation of these parameters.

DRIVE

DRIVE is used to reassign different 'logical' drive numbers to the actual 'physical' drives of your system, or display the current drive assignment, or write-protect a drive. For example, since most users prefer to work with 'drive 0' as their main drive, this allows them to use different physical drives as their 'drive 0'.

There are two ways of using DRIVE. Just a plain

```
SK*DOS: DRIVE
```

will display the current assignment. On the other hand, the command may also be of the form

```
SK*DOS: DRIVE Lx=Ty[P or U] [LD=MD]
```

which would assign logical drive x to become Type y. x and y can be numbers from 0 through 9, and T can be N for None, F for Floppy, H for Hard, O for Other, or R for RAM Disk. (There can only be one RAM disk.) Multiple assignments can be made on one line.

There are two options which can be specified, as shown in brackets above. First, any assignment can be followed by the letter P to write-protect that drive, or U to un-protect the drive. Second, the syntax LD=MD would cause DRIVE to set the MAXDRV variable equal to the last drive number currently in use, if it is not already so set.

For example, after the command

```
SK*DOS: DRIVE L0=F0 L1=H0P L2=N LD=MD
```

floppy drive 0 would become logical drive 0, hard drive 0 would become logical drive 1 (and write-protected), and logical drive 2 would be disabled. Furthermore, MAXDRV would be set to 1, assuming that there were no other active drives (set via earlier DRIVE statements or by default.)

One physical drive cannot be assigned two logical drive numbers; if you attempt to do so, DRIVE will print an error message and ignore the entire command.

Although DRIVE can be useful with floppy drives, its main use comes with hard (Winchester) disks, for it allows a hard disk to be used as logical drive 0; this is especially useful when SK*DOS is booted from a hard disk rather than a floppy. It is also essential in systems where a hard disk is partitioned into two or more 'partitions'; it then allows any combination of disk partitions to be assigned logical drive numbers, and also allows some of them to be write - protected. (If you have a hard disk, then see the HDFORMAT description for further information on hard disk partitions.)

EDLIN

EDLIN is a simple line editor for generating or modifying text files. It is not intended as a replacement for a full screen editor or word processing program; rather, it is designed to perform simple editing functions for users who do not have a more complex editor program.

In keeping with its simplicity, EDLIN has several limitations. It cannot handle a line longer than 79 characters, and it is limited to working with text files which fit wholly into memory. This may be a problem if you only have a few K of free memory, but should not be a limitation in most 68K systems. It does not provide a full screen display, but limits you to working with one line at a time.

EDLIN is called with the command

```
SK*DOS:   EDLIN <file-spec>
```

where the <file-spec> is the name of the file to be generated or modified. If not specified, the extension defaults to .TXT. If the file already exists, it will be read into EDLIN's memory and can then be edited. When you exit, EDLIN will rename the old file to an extension of .BAK (deleting an old .BAK file if it exists), and rewrite the new file with the same name as the old.

EDLIN is a *line* editor. That is, it works on lines of text. Each line of the file has a line number; these numbers are dynamic in the sense that they automatically change as new lines are added or deleted. Each time a line is printed out, its current number is printed with a colon at the left, as in

```
5: THIS LINE IS NOW LINE 5 OF THE FILE
```

At any time, you work on a so-called *current line,* which is your base of operations. You can move up or down from that line, but many operations can only be performed on this one current line.

EDLIN has two operating modes: 'command' mode and 'insert line' mode. In command mode, its prompt is a # sign, often followed by the current line number; in insert mode, its prompt is an = sign.

The command mode supports one-character editing commands such as P for Print or I for Insert. Such commands can be preceded by a line number if you wish to change to a different current line. For example, the command 10P would tell EDLIN to go to line 10 and then print it. (Do not insert a space between a command letter and any of the arguments before or after it.)

The following EDLIN commands can be used:

C - Change a string on current line

The letter C is followed by a delimiter, the old string, the same delimiter, the new string, and the same delimiter. Any character can serve as the delimiter. For example, the command `10C/Hence/Thus/` would go to line 10 and then change Hence to Thus. The command may optionally be followed by an asterisk, as in `10C/Hence/Thus/*`, which makes the command a global command, changing all occurrences of Hence to Thus from the current line down to the end of the file.

D - Delete the current line

This command deletes the current line, and renumbers all of the following lines.

F - Find a string below current line

The letter F is followed by a delimiter, the string to be found, and the same delimiter, as in `F/this string/`. The command may optionally be followed by an asterisk, as in `F/this string/*`, which makes the command a global command; it will then display every line from the current line down to the end of the file containing the specified string.

G - Go to a line

The G command is used to go to a new current line. It is followed by one of the following: line number, as in `G20`; the letter T to go to the top of the file; or the letter B, as in `GB`, to go to the bottom.

I - Insert a new line after the current line

Use the I command to insert one or more new lines after the current line by typing in an I and a carriage return (or enter). The prompt will now switch to an = sign, and you can enter one or more new lines. To switch from enter mode back to command mode, type a # sign at the beginning of a new line.

P - Print

The P command prints one or more lines, beginning at the current line. It may be followed by one of several arguments, as in these examples:

`P`	Prints the current line
`P5`	Prints 5 lines beginning with the current line
`P#5`	Prints from the current line down to line number 5
`P!`	Prints from the current line to the bottom of the file

Q - Quit without saving

The Q command exits back to SK*DOS without saving any text on the disk, and is used primarily if you wish to abandon all the work you have edited.

S - Save and then exit to SK*DOS

The S command saves the current text to the disk and then returns to SK*DOS. If, as explained above, EDLIN is modifying an existing file, then it renames the old file to .BAK before saving the new text.

? - Print a help message

Prints a brief summary of EDLIN commands.

FIND

FIND may be used to find specific occurrences of a string within a text file. The syntax is

```
SK*DOS:  FIND <file-spec> <search string>
```

For example, the command

```
SK*DOS:  FIND SKEQUATE END
```

will display the line number and text of every line which contains the string END.

FORMAT

FORMAT is used to initialize a blank disk and prepare it for use with SK*DOS. It completely erases a disk, writes the System Information Sector on the disk, initializes an empty directory, and sets up the remainder of the disk as free space.

This section describes the FORMAT command in general terms; the particular FORMAT command provided for your particular disk controller may be slightly different, but the general operation will be the same. (There may be additional information supplied with this manual describing FORMAT in greater detail.)

To format a disk, enter the command FORMAT followed by the drive number. For example,

```
SK*DOS: FORMAT 1
```

would format the disk in drive 1. FORMAT will then ask

```
HOW MANY TRACKS?
```

to which you may answer any number from 2 through 80. Be sure not to specify more tracks than your drives can handle. You will normally answer 35 or 40, but formatting fewer tracks is often convenient when you wish to save time and don't need as much space on the disk.

If your disk controller supports double-sided or double density operation, the next questions will be

```
SINGLE OR DOUBLE SIDED?
SINGLE OR DOUBLE DENSITY?
SINGLE OR DOUBLE DENSITY TRACK 0?
```

Answer with an S or D, as appropriate. (The track 0 question will only be asked if you had specified double density for the disk itself.)

You should specify a single-density track 0 only if you intend to be compatible with 6809 versions of SK*DOS; the standard SK*DOS/68K format is double density, including a double-density track 0. Although SK*DOS can read and write single-density disks, note that a "boot disk" (i.e., a disk which will be used for booting SK*DOS) **must** be double density throughout.

You will next be prompted

```
ENTER DISK NAME:
ENTER DISK NUMBER:
```

The disk name must conform to standard file-name rules (and may have an extension) and the number may go from 0 through 65535.

IMPORTANT NOTE: Although a disk name and number are not required to use a disk, you should get in the habit of putting a distinctive name and number on every disk because SK*DOS periodically checks them to make sure you have not switched a disk while a file is open on it. Giving every disk the same (or no) name and number will defeat this file protection method.

While formatting the disk, FORMAT will let you know its progress, and may display messages if errors are encountered.

Most of FORMAT's time is spent checking each sector written to make sure it is readable. Unreadable sectors are removed from the chain of free sectors, and at the end of the formatting process, FORMAT displays the number of good free sectors available on the disk. Any number of defective sectors can be thus bypassed, but formatting will be aborted if an error is discovered in several crucial sectors of track 0 (sectors 1, 2, 3, or 5).

You may also specify an optional interleave factor on the command line, as in

```
SK*DOS: FORMAT 1 5
```

which would establish an interleave of 5. Allowable values are from 1 to 9 for single-density, 1 to 17 for double density; if no interleave is specified, a default value of 3 is chosen. Changing the interleave constant may speed up some operations but slow down others; the value of 3 seems a good compromise for most uses.

(See also the LINK command description).

FROMSDOS and TOMSDOS

These two commands are used for transferring text files between SK*DOS/68K and IBM-compatible PC's running MS/PC-DOS, using the 180K disk format (one side of a standard 360K disk).

To transfer a disk between an MS/PC-DOS system and a SK*DOS system, proceed as follows:

1. First use the MS/PC-DOS FORMAT command to generate a special single-sided transfer disk. The command is

FORMAT A: /1

(which assumes that the disk is in drive A:). It is important to start with a fresh, newly-formatted disk. Then do one of the following:

2a. To transfer a text file *from* MS/PC-DOS *to* SK*DOS, simply copy it to this disk and then convert it to SK*DOS format with FROMSDOS. Note that only one file may be copied to this disk. If you have more than one file, you must use a fresh disk for each. The syntax for using FROMSDOS is

```
FROMSDOS <MS/PC-DOS drive number> <SK*DOS file spec>
```

For example, to copy a file from an MS/PC-DOS disk in drive 1 to FILE.TXT in drive 0, the command would be

```
FROMSDOS 1 FILE
```

The SK*DOS file specification defaults to the working drive and a .TXT extension, unless specified otherwise, so the file goes to 0.FILE.TXT if drive 0 is the working drive.

2b. To transfer a file *from* SK*DOS *to* MS/PC-DOS, take the disk formatted above to your PC system and run the following Basic program using the PC's Basic or GWBasic (it takes a minute or two):

```
10 OPEN "A:TEXT.TXT" FOR OUTPUT AS 1
20 PRINT #1, "GARBAGE"
30 GOTO 20
```

This initializes the disk directory in a special way. Now take the disk to your SK*DOS system and run TOMSDOS to copy the desired file to the disk. Only one file can be copied to this disk; if you have more than one, you must start with another formatted disk. The syntax for using TOMSDOS is

TOMSDOS <SK*DOS file spec> <MS/PC-DOS drive number>

For example,

```
TOMSDOS FILE 1
```

would copy 0.FILE.TXT to the previously formatted MS/PC-DOS disk in drive 1. If you get a warning message, make sure that you have specified the correct drive number for the MS-DOS disk.

The disk can now be read on your MS/PC-DOS system. If you wish to copy the file to another disk, you must use the /A option of MS/PC-DOS's COPY command. For example, to copy the file to a hard disk, the command would be

```
COPY /A A:TEXT.TXT C:
```

FTOH
HTOF

These two programs are used to do a floppy disk backup on systems which only have one floppy drive, but which have either a hard disk or a large enough RAM disk to store the entire floppy disk contents.

FTOH is used to copy an entire floppy disk (including boot sectors, directory, and even empty sectors) to a disk file on a hard disk or RAM disk. The syntax is

```
SK*DOS:  FTOH <floppy drive number> <hard disk file spec>
```

HTOF is then used to do the opposite - copy the disk contents from the hard disk or RAM disk file to the floppy disk. The syntax is

```
SK*DOS:  HTOF <hard disk file spec> <floppy drive number>
```

In both cases, only a floppy drive number is specified as the entire disk is copied, but a hard disk (or RAM disk) file name is needed since the floppy disk contents occupies only a single file on the hard disk or RAM disk.

GET & GETX

GET and GETX are memory-resident commands used to load a binary file into memory. The word GET (or GETX) is followed by the file-name of the file to be loaded. An extension of .BIN is assumed, and the current work drive is used, unless specified otherwise.

For example, the command `GET CAT.BIN` or just `GET CAT` would load a binary program called CAT.BIN from drive 0 and place it in memory.

GET and GETX usually add the current value of OFFSET (see Chapter 11) to both the load address and execution address of the program loaded. Most often, this will result in the program being loaded into memory directly above SK*DOS itself.

For special uses, the addition of OFFSET can be defeated by (a) specifying the full name and extension of the file to be loaded, and (b) immediately following it by a - (minus) sign, without a space between the extension and the minus. Note, however, that this is not normally done and may create some problems; read the discussion of the SAVE to see the implications.)

The difference between GET and GETX has to do with error checking. As GET loads a binary file, it checks the loading address of every byte. It will not allow a byte to be loaded below the current value of OFFSET, or above the current value of MEMEND; if any such invalid address is found, GET prints an error message and aborts loading. GETX, on the other hand, does no such checking. In general, you should therefore always use GET rather than GETX; GETX should be reserved only for special (and well thought out!) purposes.

HDFORMAT

HDFORMAT lets you format a hard disk for use with SK*DOS (as opposed to FORMAT, which is strictly for floppy disks). This command will work slightly differently for different systems, and you may have received an addendum which describes any such differences for your system configuration. (The following describes the command as used with the WD1002 controller.)

HDFORMAT is called with the simple command line

```
SK*DOS: HDFORMAT
```

and then proceeds to ask for the information it needs to properly format the hard disk. It's important to answer these correctly, and so we will discuss these questions and their answers at length. The first question is

```
Which drive - A or B?
```

In most instances, only one drive is present and so the choice is A. (The WD1002-HDO controller used on some computers allows three hard drives, which it calls drives 0, 1, and 2, but SK*DOS supports only drives "1" and "2". To avoid confusion with drive numbers under SK*DOS we therefore call these two drives A and B instead of 1 and 2. SK*DOS does not allow what the WD-1002-HDO calls "drive 0" because the WD-1002-HDO switches to this drive number upon powering up, and not putting an actual drive on this port avoids the possibility of the disk being corrupted when powering up or down.)

The next four questions have to do with the size and configuration of the disk, which requires some explanation first.

Hard ("Winchester") drives typically contain several 'platters' which rotate on a common shaft. Each platter has two sides, with a head on each side to read and write data. The heads are all mounted on a common assembly so they move in and out together. At any one time, each head is positioned over a circular track on its surface. All of those tracks together make up a 'cylinder'. The data on each track is divided into 'sectors', with typically thirty two 256-byte sectors per track; the number of sectors per cylinder is then equal to the number per track times the number of tracks per cylinder.

For example, a MiniScribe 3425 drive has two platters with four heads. There are 615 tracks on each platter (so there are 615 cylinders) and 32 sectors on each track. Beyond the 615th track, however, this drive has additional tracks which, though not good for storing data, are specially set up as a 'parking area' - an area designed for letting the heads land when power is shut off. The drive instructions specify that the parking area is on cylinder 656.

Winchester drives also use 'write precompensation' to reduce write errors. Precompensation is only needed on inner tracks, and different drives need it on different tracks. Many drives start precompensation on track 128, but you should consult the drive manual for the correct number to enter.

Using this information, the next four questions (and their answers for this sample drive) would be

```
Number of cylinders (Tracks/Side) to use on drive? 615
Which cylinder should the drive be parked on? 656
Which cylinder to start precompensation? 128
How many heads on the drive? 4
(There are 32 sectors per track/side.)
```

The park cylinder is not actually used by HDFORMAT; instead, it is saved and used by the PARK command.) Even when a specific parking area is not suggested by the disk manufacturer, we recommend that you reserve some cylinders for that purpose, and subtract that number of cylinders from the total capacity of the drive.

In some cases, you may want to use only part of a hard disk for SK*DOS, leaving the remainder for some other DOS. In that case, simply specify either a smaller number of heads or a smaller number of cylinders. For example, you could specify 309 cylinders, which would leave the other 306 for use by the other DOS.

SK*DOS next asks

```
Enter the required code for drive step rate -
   0 if buffered, 1 to 15 for 0.5 to 7.5 msec respectively:
```

The step rate determines how fast the drive will move the heads from one track to the next. In the buffered mode, the controller tells the drive to move as fast as it can, and simply waits until the head movement is finished; in the other modes, stepping signals are sent at fixed intervals from 0.5 to 7.5 milliseconds apart, until the drive is at the requested cylinder. You will have to consult your disk drive manual to determine its capabilities. Like most modern drives, the MiniScribe drive described above is capable of either buffered stepping, or any fixed rate of 3 msec or more. Hence you may enter either a 0 for buffered, or a 6 for 3.0 msec stepping (the above code numbers are specified by the disk controller, and the number to enter is equal to twice the speed in milliseconds.) Given the choice, buffered stepping will always be faster.

HDFORMAT will now use the data you have entered and compute the total capacity of the disk as specified. If you have entered a smaller number of heads or cylinders because you wish to reserve part of the disk for use by another DOS, then only the usable part of the disk will be used in this calculation. For the above example, HDFORMAT would print out

```
Total disk capacity is 19680 K
```

which is a bit over 19 megabytes.

Because of the structure of SK*DOS, each logical drive is at this time limited to a maximum of 16 megabytes, so the above example shows that this particular Winchester drive has more capacity than SK*DOS can handle as a single drive. It will therefore print out the message

```
THIS EXCEEDS THE MAXIMUM LOGICAL DRIVE CAPACITY; you must
therefore partition the disk into 2 to 4 partitions.
Would you like to partition the disk (Y/N)?
```

In order to use the full 19+ megabyte disk, you must partition it into at least two partitions. Each of these partitions becomes a separate logical drive, and there is a maximum of four partitions allowed per hard disk. (Even if a drive does not exceed 16 megabytes, you may still partition it even though you do not need to; in our particular example you **must** answer Y to the last question since the drive is just too big for one partition.) Assuming a Y answer, HDFORMAT then asks

```
How many partitions would you like?
```

Still assuming our previous example, suppose you answer 3 to make three partitions. Then the next series of messages is

```
You will now partition the disk by Cylinders (Tracks). The
    TOTAL number of usable cylinders on this drive is 615.
```

```
The MAXIMUM number of cylinders allowed in any one
    partition is 512.
The MINIMUM number of cylinders allowed in any one
    partition is 5.
Decide how you would like to split up the TOTAL number of
   cylinders and enter the counts:
Cylinders for Partition H0: 205
Cylinders for Partition H1: 205
Cylinders for Partition H2: 205
```

In this example we split the disk evenly by assigning 205 cylinders to each partition. Although there is no need to make the partitions equal, it may be convenient if you intend to use BACKUP to quickly copy one partition to another.

Assuming no further errors, the next message will be

```
READY TO FORMAT HARD DRIVE
ARE YOU SURE YOU WANT TO GO AHEAD?
```

which should be answered with Y.

There are a number of self-explanatory error messages which may come up during the above procedure. For example, selecting a partition size that is too small or too large, or selecting partition sizes which do not add up to the total number of cylinders on the disk, will give an error message and allow you to re-enter the data.

Once formatting starts, HDFORMAT will print out status information so you can see what it is doing. If the disk has defective sectors, their locations will be displayed and they will be deleted from the free space on the disk. A relatively large number of defective sectors can be tolerated on a disk without a problem, except that logical track 0 sectors 0, 3, and 5 must always be good on every partition - errors in any of these will cause an immediate "Fatal Error" message and HDFORMAT will quit. If this occurs in partition 0 of a disk, then the hard disk is not usable. If it occurs in another partition, however, then it will generally be possible to move these defective sectors to some other track by partitioning the disk differently, and then reformat the disk without problem.

Different disk partitions can be assigned different logical drive numbers with the DRIVE command. Partitions are called H0 through H7, as follows:

Drive A has H0 through H3
Drive B has H4 through H7

Finally, some notes about how HDFORMAT works. Most hard disks arrive from the factory with one or more defects; on a brand new disk drive these will generally be carefully documented on a test printout enclosed with the drive. Most such errors involve just a few bits on isolated tracks; because most modern hard disk controllers use error correction, they are able to correct such minor errors and therefore hide them. As HDFORMAT verifies the disk, however, it writes and then reads each sector. If the sector was not read correctly, then it is obviously bad; even when read correctly, however, HDFORMAT still checks whether a sector required error correction. If so, HDFORMAT tries to read that sector once more. If it still requires error correction (even though it may have been read correctly both times), HDFORMAT still flags that sector as defective. This will hopefully avoid future errors.

You may note that the number of free sectors displayed in a directory printout is always lower than the number calculated from the number of cylinders. Part of this is due, of course, to the fact that the free sector printout does not include those in the directory and system information sector. In addition, HDFORMAT always formats the disk

so that each track has the same number of sectors. In some cases the partitions are such that HDFORMAT has to discard a few sectors so as to make the logical tracks all the same length.

Finally, we leave the space below so you can record the characteristics for your hard drive(s) to make future reformatting easier.

Drive Manufacturer and Type	Number of Cylinders	Park Cylinder	Start Precomp on Cyl.	Number of Heads	Sectors per Track	Step Rate Code

HELP

HELP is simple program which merely reminds you how to get help with other SK*DOS commands - simply type their name, a space, and a question mark.

LINK

LINK is one of the commands used to generate a 'bootable' SK*DOS system disk on some systems. It tells the super-boot program where to find the SK*DOS program on the disk.

Here is a short explanation of why LINK is needed. To start SK*DOS, you usually use a monitor command. In systems which contain HUMBUG or another customized monitor, this command may boot SK*DOS directly. In simpler systems, however, this monitor command may only be capable of reading one sector from a disk. In that case, it loads a 'super-boot' program from track 0 sector 1 of the disk into memory. The super-boot program then reads the rest of SK*DOS and executes it. Since SK*DOS could be anywhere on the disk, the super-boot needs a quick way of finding it. This is accomplished by using LINK to store the disk address of SK*DOS directly into the super-boot program itself.

LINK is only necessary when you want to generate a 'bootable' system disk; data disks which do not contain a copy of SK*DOS and which will never be booted do not need LINK.

There are three basic ways of generating a bootable system disk:

1. Use BACKUP to do a mirror-image copy of an existing bootable system disk onto another formatted disk. Since BACKUP does an exact copy of a disk, the resulting disk will be exactly like the original and will also be bootable. (But if the original disk has unneeded files or defects, so will the new disk.)

2. Use HTOF to restore a disk image (previously stored with FTOH) of a bootable disk onto a floppy disk.

3. Use FORMAT to initialize a blank disk, then use COPY to copy SK*DOS to it, and then use LINK to give its address to the super-boot program. In this way it is possible to copy only needed files to the new disk. Use of COPY instead of BACKUP also compacts the disk and rearranges the files to make disk access faster.

To link a disk, enter the command LINK followed by the SK*DOS file name. For example,

```
SK*DOS: LINK 1.SK*DOS.SYS
```

would tell the super-boot in drive 1 where to find the file SK*DOS.SYS. You must enter the extension. The drive number is optional, and will default to the working drive if omitted.

Incidentally, self-contained programs (those which use the monitor but not SK*DOS itself) can also be linked to the super-boot. In that case, they will be loaded and executed directly upon booting the system.

LIST

The LIST command will list the contents of a disk text file to the screen or printer.

The LIST command, in turn, requires the name of a file to list. It assumes that the required file name has been entered on the same line, separated from the word LIST by either a comma or a space. For example,

```
SK*DOS: LIST TEXT
```

would list the contents of the file called TEXT. Since LIST is intended for listing text files, it assumes a .TXT extension unless told otherwise. Hence `LIST TEXT` means the same as `LIST TEXT.TXT`.

LOCATE

LOCATE is used to identify the memory locations into which a binary file will load, and to provide the transfer (or starting) address.

To use it, type the command LOCATE, followed by the name of the file desired, as in

```
SK*DOS: LOCATE FILE.COM <->
```

The file name defaults to a .BIN extension if not supplied.

The above example shows an optional - (minus) sign at the end of the command line. This affects the displayed addresses as follows:

If the - is omitted (the default), LOCATE will print the message `ADDRESSES, AS IF LOADED WITH CURRENT OFFSET:` and display the actual addresses the file would use if loaded or executed with the current OFFSET. (These addresses consist of the addresses stored as part of the disk file, plus the current value of OFFSET.)

If the - is present, LOCATE will print the message `ADDRESSES, AS SAVED ON DISK WITHOUT OFFSET` and will display the disk file addresses as they exist on the disk file, without adding the current value of OFFSET. This option describes the file, but does not provide the information as to where the file would load in the current system.

Note that LOCATE's printout does not necessarily specify all of the memory used by a particular program, since the program may use memory areas without actually loading anything into them from disk.

When several binary programs are appended together, they may contain more than one transfer address. Although LOCATE will indicate them separately, keep in mind that only the last transfer address will actually be used by SK*DOS.

MAKEMPTY

MAKEMPTY is a command which generates an empty file.

Perhaps a bit of an explanation is in order. SK*DOS does not normally like empty files. When a program asks SK*DOS to open a file for writing, but does not actually write anything into it, SK*DOS does not actually put the file on the disk. In other words, it will not generally create a real, but empty, file on the disk (unless you reset the computer or remove a disk while a file is open for writing, which is something you should **never** do!)

Nevertheless, there are times when an empty file would be useful - for example, to initialize a data file which is to be read by a Basic program, updated with more data, and then rewritten. That is precisely what MAKEMPTY does.

To use MAKEMPTY, follow the command with a file name, as in

```
SK*DOS: MAKEMPTY DATAFILE
```

This command would create an empty file called DATAFILE.DAT on your working drive, since a default extension of .DAT is assumed unless specified otherwise.

The resulting empty file contains all zeroes, which is interpreted by Basic as well as text editors as truly empty; an attempt to read this file will result in an immediate end-of-file indication.

MON & RESET

MON and RESET are used to exit SK*DOS and return back to the system monitor (if there is one in your computer system). Both of these are memory-resident commands.

Not all systems have both MON and RESET; sometimes only one or the other may be available. The difference between them has to do with their treatment of exception vectors (traps).

When running, SK*DOS always resets the "line 1010" exception vector of the 68xxx CPU to its own value. In addition, depending on the state of TRPFLG, SK*DOS may also reset all of the other exception vectors as well. If both MON and RESET exist on a system, then

`MON` will return to the monitor at a point where the ROM monitor will leave the exception vectors as they were set up by SK*DOS.

`RESET` will return to the monitor at a point where the monitor will do a complete reset, including reinitializing all of the exception vectors.

PARK

The PARK command is used for parking the heads of hard drives (Winchester drives). It is called simply with

```
SK*DOS: PARK
```

and takes its parameters from the data stored on the disk during formatting. Parking the heads of a hard disk drive moves them to a section of the disk which is not used, and prevents possible damage to the usable portion of the disk surface or to the data stored on it.

Parking the heads before moving a disk drive is generally regarded as desirable, although there is some disagreement among users as to whether parking heads is necessary when just shutting the system off. Nevertheless, it is our opinion that devoting a part of the disk to a parking area and then always parking the heads in that area before shutting off the power is a sensible precaution, despite the fact that it may slightly reduce the disk capacity.

The literature accompanying some drives specifies a park cylinder to be used; if so, then the choice is simple. In other cases, however, the choice may not be so clear.

PDELETE

The PDELETE command is used to rapidly delete one or more files from a disk. Its advantage over DELETE is that you need not type in individual file names, since PDELETE will prompt you with the file names.

The syntax for using PDELETE is

```
SK*DOS:  PDELETE <drive number & directory> <optional match list>
```

The first parameter must include a drive number and may also include a directory name. If only a drive number is specified, then the current default directory of that drive is used; if both a drive number and directory are specified (such as 1.D/) then only the files in that directory may be deleted. An asterisk (*) used as the directory (as in 1.*/) would refer to *all* directories of the specified drive.

The match list works like that of CAT, COPY, or TCAT. If not given, then PDELETE will prompt with the names of all the files on the specified disk or directory, one at a time. If given, then only files which match the match list will be prompted. For example, the command

```
SK*DOS:  PDELETE 1 TX .CMD
```

would prompt with the names of all files whose names begin with TX, or which have .CMD extensions. In each case, you will be asked whether to delete, and can answer Y (yes), N (no), or Q (quit).

PEEK
POKE

The PEEK and POKE commands are similar to the same commands of Basic. These commands can be used to make minor modifications to SK*DOS or other programs.

To use PEEK, give the command `PEEK` followed by a hexadecimal address. The contents of that address will be printed out.

To use POKE, enter the command `POKE` followed by the hexadecimal address of the location to be poked and the hexadecimal number to be poked into that location. The POKE command will print out both the old and the new contents as a check.

When PEEKing or POKEing non-existent memory locations, you may get undesirable results. On some systems (which do not generate bus errors), the system may totally die. If the hardware properly generates bus errors, then you will get an error message, either from SK*DOS (if TRPFLG is non-zero so that SK*DOS handles all exceptions), or from the monitor (if TRPFLG is zero so exceptions are handled by the monitor).

PICTEST.BAS

PICTEST.BAS is a Basic program which helps to check that an assembly language program written for use with SK*DOS is indeed relocatable. The version supplied is specifically written for use with TSC XBASIC on 6809 systems used for development, but can be easily modified for use with other Basic interpreters.

Although it is somewhat difficult to test an assembly language source file for position independent (relocatable) coding, it is easy to test the machine language object code. The trick is to simply assemble the same program twice, once with an ORG of $0000, and once with a different ORG (such as $1000). If the program is truly relocatable, then the two object codes should be exactly identical (although certain programming techniques may result in slight differences.) Then it is only necessary to compare the two object files for differences. PICTEST.BAS does this job by reading the Motorola S1-type files output by an assembler such as TSC's 68000 Cross-assembler.

Before using PICTEST.BAS, assemble the program to be tested with two different ORG statements as described above. Then execute your Basic interpreter and load PICTEST.BAS. When it is executed, PICTEST.BAS will ask for the names of the two binary files to be compared. If any differences are found between the two programs, PICTEST.BAS will print out the address where the difference was found.

Once such a list of addresses is found, you should check these against a listing of the program being tested. Some differences may be valid, depending on your programming style; others may be caused by errors, such as perhaps forgetting to use BSR instead of JSR, or forgetting to use (PC) after a variable name.

PROMPT

PROMPT allows you to change the system prompt from the normal SK*DOS: to any other string. For example, users who feel more at home with a shorter prompt such as, perhaps, three plus signs, can change the prompt with the command

```
SK*DOS: PROMPT +++
```

The new prompt may be up to 10 characters long, and may include any printable character.

PROMPT only changes the prompt currently in memory - it does not change the prompt string on the boot disk. Hence the original SK*DOS: prompt will return the next time you boot the system. Moreover, PROMPT may only be used once; that is, once PROMPT changes the current prompt it cannot change it again unless you reboot the system.

PROTECT

Files may be assigned one or more kinds of protection codes as follows:

a. CATALOG protected files will not appear in a catalog or directory listing (although the commands supplied with SK*DOS ignore this kind of protection.)
b. DELETE protected files cannot be deleted.
c. WRITE protected files cannot be written over.

The PROTECT command is used to assign protection codes to a file. To use PROTECT, give the command PROTECT, followed by the file name, followed by one or more of the following codes:

C = Catalog protect
D = Delete protect
W = Write protect
X = Cancel protection.

For example, the command

```
PROTECT SK*DOS.SYS WD
```

would prevent SK*DOS.SYS from being deleted or written over.

```
PROTECT LIST.COM XD
```

would cancel whatever protection LIST.COM currently has and instead substitute delete protection.

If protection codes contradict each other, the rightmost codes take precedence. For example, the code CDWXC would assign catalog, delete and write protection, then cancel them all, and instead provide only catalog protection.

RAMDISK

RAMDISK allows you to set aside a fixed area of your RAM memory as a RAM disk (also called a virtual disk.) The RAM disk is an area of RAM which is used exactly like a real (or physical) disk drive. It has a drive number, is accessed exactly the same as a real disk, has a directory like a real disk, and can store programs or data files like a real disk. The main disadvantage of a RAM disk as compared with a real disk is that the RAM disk becomes erased when power is turned off (or if there is a power outage.) Hence any important data on a RAM disk should be copied to a real disk before power is turned off, and perhaps at frequent intervals between.

RAM disk is enabled with the command

```
SK*DOS: RAMDISK <drive number> [<RAM disk size>]
```

where

the <drive number> may be any number from 0 through 9, which then becomes the drive number of the RAM disk. If the drive number coincides with that of a real disk drive, then the real drive becomes inactive and the RAM disk takes its place. (A drive number must be supplied.) If the RAMDISK drive number is larger than any other existing drive, the value of MAXDRV will automatically be changed to reflect the RAM disk drive number. Since most systems will not have the maximum number of ten drives allowed by SK*DOS, it will usually be possible to place the RAM disk above other real drive numbers.

the <RAM disk size> is the size, in increments of 4K, of the memory to be devoted to the RAM disk, with a minimum of 8K. If the amount entered is not a multiple of 4K, then the next lower 4K multiple will be used. It can be entered either as just a plain number (such as 16) or followed by the K (as in 16K). When RAMDISK is first called, a size must be specified, but no size need be entered if the RAMDISK command is used to reassign the drive number of an existing RAM disk. When initializing a new RAM disk, the RAMDISK program will calculate the amount of user memory left, and abort if less than 16K would be left. Otherwise, it will display the amount left and ask whether to proceed. If you do not answer with a Y, it will abort without setting up the RAM disk. Make sure not to allocate so much RAM to the RAM disk that not enough is left for other uses.

While RAMDISK is initializing, it will print out a row of periods, one for each "track" being initialized. Note that the RAMDISK program checks for the existence of enough memory, but does not check whether that memory is working correctly.

Once the RAM disk has been initialized, it may be reassigned to a different drive number by again typing the RAMDISK command, followed by the new drive number (but without a size). Its size, however, cannot be changed.

Both the RAMDISK program, as well as the extended disk memory, contain flags which indicate the status of the RAM disk. If you reboot SK*DOS (without powering down the system), the newly booted SK*DOS will not know about the RAM disk, but the RAM disk contents in memory will be retained as long as it is not overwritten with other data. If you immediately enter the RAMDISK command, the RAMDISK program will reenable the RAM disk, but will print the message `RAM DISK WAS FORMATTED EARLIER` rather than erasing it and formatting it again. Hence your data will be retained in the RAM disk (still assuming, of course, that power has not been turned off in the meantime or that the RAM disk memory has not been overwritten with other data.)

If you do wish to actually erase all data on the RAM disk, insert the word NEW into the command, as in

```
SK*DOS: RAMDISK NEW x
```

which will totally erase the RAM disk and then reformat it as drive x.

RAMDISK changes both the OFFSET and MEMEND pointers. The RAMDISK program itself is loaded beginning at the current value of OFFSET, and then OFFSET and OFFINI are changed to point above the program so that subsequent programs are loaded above the RAMDISK program. The virtual disk memory itself, into which RAM disk data is stored, is placed at the top of user memory just under the initial value of MEMEND, and MEMEND is then reset to point just under it.

This has a bearing on what happens if SK*DOS is exited and rebooted while a RAM disk exists. After rebooting, the RAMDISK program will no longer exist in memory, but the RAM disk data should still exist at the top of memory *below* the current value of MEMEND. As long as no program is run which uses more than 16K of RAM, this data will continue to exist and the RAMDISK can be reinstated by another RAMDISK command without data being lost.

REDOFREE

REDOFREE is used to rearrange the sequence of the free sectors of a disk so they go in order from the outside of the disk toward the inside. The sectors are in this sequence on a fresh disk, but gradually become disordered as a disk is used because each time that a file is deleted, its sectors are added to the end of the free space. The space from deleted files is thus scattered around the disk, with the result that the free space also wanders all around the disk.

The syntax for using REDOFREE is

```
SK*DOS:   REDOFREE <drive number>
```

REDOFREE then prints a number of basic parameters about the disk, reads all the sectors in the free space, sorts them into numeric order, and finally rewrites the sector linkages to place them into numeric order. (After each of these steps, REDOFREE asks whether to proceed, so you may stop it at any point.)

REDOFREE provides the option of printing a listing of free sector segments both before and after they are sorted. If the listing is selected, each line may be followed by one of two marks:

1. Before the segments are sorted, a caret mark indicates that the current segment points backward, and thus the free chain goes backward. This will be corrected after sorting.

2. After the segments are sorted, a comma indicates that the current segment is contiguous to the next, and that these two segments will be combined into one if the free space is relinked.

RENAME

RENAME is used to rename a disk file. To use this command, enter the word RENAME, followed by the old name and the new name, as in

```
RENAME OLDNAME.TXT NEWNAME.ABC
```

No quotes are needed, but the extension must be supplied for both names.

Optional drive numbers and/or directory names may be supplied for one or both files, as in

```
RENAME 1.F/OLDNAME.TXT 1.A/NEWNAME.TXT
```

If not supplied for the old file, the drive and directory names default to the current work drive and directory. If not supplied for the new file, they are the same as the old file.

S1TOCOM

S1TOCOM is a program which converts a binary file in Motorola S1-S9 format into a binary file in SK*DOS format. It is used to convert the output of some 68000 assemblers into a format which can be loaded and executed by SK*DOS.

S1TOCOM is supplied in two forms - S1TOCOM.COM is the 68K version, while S1TOCOM.CMD is a 6809 version intended for systems programmers who may be using a 6809 system for software development.

The syntax for using S1TOCOM is

```
SK*DOS: S1TOCOM <file name> [-]
```

where the file name is the name of the binary S1 file to convert. The input file defaults to a .BIN extension, while the output file will have the same name but with a .COM extension. If a .COM file already exists, S1TOCOM will delete it without any prompt; after it is finished, S1TOCOM will delete the input file, again without any prompt.

If the optional "-" sign does not follow the file name, then the loading and transfer addresses in the output file will be exactly the same as those in the input file. If the "-" sign is entered, then the current value of OFFSET will be subtracted from the addresses given in the input file. The "-" option is primarily for programmers who wish to convert a non-position independent file to SK*DOS operation, and have assembled that file to lie in a memory location known to be available. But note that this option can lead to great problems if the value of OFFSET at the time the conversion to binary is done is not the same as the value when the converted program is executed. It would be much safer to use S1TOCOM without the "-" option, and then use the "-" option of GET, followed by XEQ to execute the final program.

SAVE

This memory-resident command saves binary data from memory to disk. For example, the command

```
SAVE PROGRAM 9000 A000 9004
```

would save a file called PROGRAM from memory locations $9000 through $A000, and assign a transfer (starting) address of $9004.

All addresses are hexadecimal, the file name extension defaults to .BIN if not given, and the transfer address is omitted if not given.

Once called, the SAVE command asks the following question:

```
SAVE OPTIONS:
  1. USE ABOVE ADDRESSES (IGNORING OFFSET), OR
  2. SUBTRACT OFFSET FROM ABOVE ADDRESSES
CHOOSE 1 OR 2:
```

These options affect only the addresses stored on the disk as part of the file, not the address of the memory area saved to disk, as follows:

Option 1 - Use Above Addresses. The addresses specified (9000, A000, and 9004 in the above example) are stored on the disk as the load and transfer addresses. In other words, the addresses saved as part of the file correspond exactly with the area of memory saved to disk. In general, however, if this file is subsequently loaded back into memory, it will usually go back into a different location, since most loading is done by adding the current value of OFFSET to the addresses actually stored as part of the disk file. If you desire to load this file back into exactly the same memory locations it came from, you must use the - (minus) option of GET.

Option 2 - Subtract OFFSET From Above Addresses. In this case, the current value of OFFSET is subtracted from the specified addresses before they are written to the disk. For example, if the current OFFSET is $8000, then the file stored in the above example would contain addresses 1000, 2000, and 1004. When loaded back into memory with GET, however, these addresses would be added back to OFFSET, so that the file would go back into the same memory area it was saved from (assuming OFFSET remained the same). Option 2 cannot be used, however, if OFFSET is larger than any of the specified addresses; in this case SAVE will print an error message and abort before finishing the file.

You should try saving a few files and then examining them with LOCATE to get a feeling for these two options.

SCAT & SEQUENCE

SCAT is similar to CAT, but differs in three ways:

(1) SCAT prints both the date of each file as well as its sequence number (if one has been assigned to the file),

(2) The catalog listing is sorted by date and sequence number so that the most recent files appear first,

(3) SCAT does not allow all of the options used by the CAT command.

SCAT is designed for systems which do not have a clock IC installed. (If you do have a clock, then you should read the description of the TCAT command which, along with time stamping, provides greater capabilities.) When a clock is not installed and interfaced with SK*DOS, as described under TCAT, then SK*DOS automatically assigns a 'sequence number' to each file as it is written or updated. This number starts with 1 when the system is booted, and increments by 1 each time a new file is written. When your disk contains a number of files all having the same date, the sequence number tells you what order the files were written in. The SCAT command then prints the catalog, but sorted by date and sequence number so that the most recent file appears on top.

Obviously, if your system has a clock, then using time-stamping and TCAT provides the actual time rather than just a sequence number, but the sequence number is a good second best if no clock is available. It has three characteristics you should keep in mind:

1. Each time the system is powered up, the sequence number starts with 1. In other words, if you turn off the power and then restart, the sequence number will return back to 1. To avoid having multiple files with the same date and sequence number, you may want to get in the habit of manually updating the sequence number if you restart the system, so that it continues with the next higher number. The SEQUENCE command is used for this purpose. For example, to change the number to 7, use the command

```
SK*DOS: SEQUENCE 7
```

Typing just `SEQUENCE` without a number will print out the current sequence number.

2. The sequence number is incremented by 1 for every file written or updated. This includes files being copied, even though copied files are not assigned a new number. In other words, each time you copy a file, the sequence will appear to skip a number; this is the number that would have been assigned to the copied file if it had not kept its old number.

3. Since the sequence number is just a single byte, the maximum number is 255, after which it returns to 0.

SCAT arguments are the same as those for ACAT.

SK*DOS09

SK*DOS09 is 6809 SK*DOS along with a 6809 CPU simulator which allows you to run 6809 SK*DOS programs on your 68K computer. This allows you to run programs which may not be available for the 68000, though at reduced speed.

To run SK*DOS09, simply type the command

```
SK*DOS:   SK*DOS09
```

As soon as the program runs, you will be greeted by the familiar SK*DOS signon, except that this time it will be 6809 SK*DOS that is running on your computer and the prompt will change to SK*D09:

Since your 68K computer is essentially interpreting 6809 instructions, you will notice a slight slowdown in operation. On input and output (such as listing a disk directory) you will hardly notice a difference, whereas on heavy computing there may be a significant slowdown, depending on the program being run. Even so, you may want to do certain operations (such as copying a disk) in 68K mode since that will always be faster.

SK*DOS09 will run most 6809 software, although there may be some programs which do not work. This will generally include those which access I/O hardware or other machine parameters directly, bypassing DOS. This includes, for example, programs which require interrupts, such as some text editors with type-ahead buffers. It also means that you cannot use a 6809 FORMAT program to format the disk - you must use the 68K FORMAT instead. Furthermore, although you might be tempted to use the 6809 SK*DOS's TTY command, don't bother; I/O is handled by the 68K SK*DOS, and so use its IOCONFIG instead. There may be other programs which don't run as well - programs which use some particular characteristic of the 6809 CPU which is difficult or impossible to simulate on the 68K CPU. If you do run across some such example, let us know and we will try to suggest some other approach.

While running SK*DOS09, the ESCape key works normally to temporarily pause output; if you follow it with CR (or RETURN or ENTER, whichever it is called on your terminal), you will return to either SK*DOS09 or, in some cases, to the application program running under SK*DOS09.

Just as the MON command in normal SK*DOS/68K brings you 'back out one level' to the program which called it (namely the monitor or boot ROM), so MON in SK*DOS09 brings you out one level to the 68K SK*DOS when you want to exit.

Finally, a few technical details. SK*DOS09 requires a minimum 128K system, and devotes the entire second 64K of that system, from $10000 to $1FFFF, to simulating the 6809 environment. Each location of that 64K 'pseudo-memory' corresponds to the equivalent location of a normal 6809 computer system. (For example, the warm start location of 6809 SK*DOS is $CD03, while in SK*DOS09 it is located at $1CD03. The initial "1" is, however, totally transparent to 6809 software, which simply addresses that location as $CD03 as it always has done since the address translation is done entirely by SK*DOS09.) Just as in a true 6809 system, though, only the first 56K is actually used; the last 8K, which would normally be taken up by 6809 I/O and monitor, is empty and available RAM.

In addition, SK*DOS09 also requires approximately 8K of user memory for its 6809 interpreter program. This normally starts at the current value of OFFSET.

SK*DOS09 uses your normal 68K I/O devices, and can access all your 68K disk drives, including RAM disk, although you must make sure that your RAM disk memory does not conflict with the memory used by

SK*DOS09. If it does, then SK*DOS09 will print an error message when it is started, indicating that there is not enough memory for it to function.

Since the disk formats for both 6809 and 68K SK*DOS versions are identical, 6809 and 68K files can coexist on the same disk. You will note, however, that 68K command files have the extension .COM, whereas 6809 command files use the .CMD extension. Hence typing a common command such as CAT or DELETE will always call the correct program. (Incidentally, both SK*DOSes will use the same ERRCODES.SYS file, since the error numbers are identical. Further, 6809 SK*DOS ignores .BAT files.)

Finally, SK*DOS09 can only access root directories (i.e., unnamed directories) of drives, since multiple directories were not available on 6809 SK*DOS systems.

STEPRATE

The STEPRATE command is used to set the speed at which SK*DOS commands disk drives to step from track to track. (Depending on your disk controller and disk drivers, this command may not exist, or may be slightly different.

For 5-1/4" disk drives, the allowable step rates are 6, 12, 20, or 30 milliseconds (ms) per step (with most floppy disk controllers), or 2, 3, 12, or 20 ms (with the Western Digital 1772 controller). The step rate is specified after the word STEPRATE as in

```
SK*DOS: STEPRATE 30
```

Each time STEPRATE is called it responds by displaying the new step rate setting; if no step rate is specified in the command or an invalid number is specified, then the step rate remains the same and the current value is displayed.

(For those users more at ease with programming floppy disk controllers, the values 0 through 3 {which are the values actually given to the floppy disk controller} may also be specified as follows:
0 = 6 ms
1 = 12 ms
2 = 20 ms
3 = 30 ms.)

Smaller step rates will produce faster operation, but you should be careful not to specify a value smaller than your disk drives can handle since that may lead to errors. If in doubt, it is better to err on the conservative side.

SYSTEM
WORK

The SYSTEM and WORK commands are used to specify the system and working drives and directories. When SK*DOS is first booted, the root directory of drive 0 becomes the default system and working drive; that is, all programs will be loaded from this system drive/directory, and all files being worked on will be loaded from this working drive/directory. The default drive and directory selection can be changed with these two commands. For example, the command

```
SK*DOS: SYSTEM 1
```

would make drive 1 the default system drive, while

```
SK*DOS: WORK 8.C/
```

would make drive 8, directory C, the default working drive and directory.

Entering the SYSTEM or WORK command without a valid drive number following causes the current drive and directory to be displayed. Entering a drive number without a directory name switches to the root directory of that drive; it is not possible to enter a directory without a drive number to go with it.

In addition, the system drive (not the work drive) can also be specified as SYSTEM ALL. In this case, SK*DOS will search the root directories of all drives, in numeric order starting with drive 0, until it finds the requested file. If SYSTEM ALL is used, it is important to make sure that MAXDRV is properly set, so that SK*DOS does not try to search drives which do not exist. Use the DOSPARAM command to inspect or change MAXDRV.

TCAT

TCAT is similar to CAT, but differs in three ways:

(1) TCAT prints both the date of each file as well as the time (if one has been assigned to the file),

(2) The catalog listing is sorted by date and time so that the most recent files appear first,

(3) TCAT does not allow all of the options used by the regular CAT command.

Although TCAT is provided primarily for those users who have a clock/calendar installed, it is also useful to others because of its ability to sort directory entries by date to display the latest files first.

Here are a few examples of how to use TCAT:

TCAT	prints a catalog of the default directory of the work disk
TCAT 5	prints a catalog of the root directory of drive 5
TCAT 5 FS	prints a catalog of all files on drive 5 which have names beginning with FS
TCAT 5.A/ FS	prints a catalog of all files on drive 5 subdirectory A which have names beginning with FS
TCAT 5.*/ FS	prints a catalog of all files in all subdirectories on drive 5 which have names beginning with FS
TCAT 5 FS.T .COM	prints a catalog of all files on drive 5 which have file names beginning with FS and also have extensions beginning with .T, and of all files on drive 5 which have the extension .COM
TCAT 5 F*S.T .COM	prints a catalog of all files on drive 5 which have file names beginning with F*S (where * stands for any character) and also have extensions beginning with .T, AND of all files on drive 5 which have the extension .COM

The possible options for TCAT are the same as for ACAT and SCAT. (Try TCAT on the SK*DOS disk to see what it does.)

If your system has a clock/calendar IC but your version of SK*DOS does not already have time stamping implemented, information on adding time stamping is provided in the 68K SK*DOS Configuration Manual.

SK*DOS's COPY command carries the old date and time along with the file. Hence copies of your files will have the same date and time as the original.

The time code is based on a 24-hour clock, and is given in tenths of an hour, ranging from 00 to 239. For example, a code of 123 means 12.3 hours, or 3/10 of an hour (18 minutes) past noon. The TCAT program would display this as 12:18. If the time code is 00, which is the case for files without a time, then TCAT will omit the time printout.

TIME

If your system has a clock/calendar IC, then the TIME utility may be supplied with your SK*DOS. It allows you to read and set this clock.

The command

```
SK*DOS: TIME
```

is used to display the current time in the format

```
DAY MM-DD-YY HH:MM:SS AM
```

The command

```
SK*DOS: TIME S
```

is used when you want to set the clock. Simply answer the questions asked to set the clock, making sure to use two-digit numbers as requested.

TRACE***
TRACENAB

TRACE*** is a memory-resident command which allows you to enter a user program with the CPU in trace mode, so that the program can be debugged. This mode requires a monitor ROM, such as HUMBUG, which supports the trace mode.

The TRACE*** command (which has three asterisks as part of the name to avoid the possibility of its being called accidentally) is used just before the command which loads and executes the program being debugged. For example, to debug a program called NEWPROG.CMD, the commands would be

```
SK*DOS:  TRACE***
SK*DOS:  NEWPROG
```

The TRACENAB command is needed only when tracing with HUMBUG and no previous tracing has been done since turning on the power. When HUMBUG traces a program, it saves the contents of the 68xxx processor's trace trap, places its own trap address into the trap, and then goes to a user program; after returning, it replaces the original address in the trap. When tracing is begun with SK*DOS's TRACE*** command rather than via HUMBUG, HUMBUG's trace routines are entered in the middle and hence HUMBUG does not have a chance to save the previous contents of the trap location. It therefore restores the wrong value into the trap location. TRACENAB corrects that problem. The correct syntax is

```
SK*DOS:  TRACENAB HUMBUG
```

UBASIC

UBASIC is not intended to replace a full-featured Basic, but does suffice for some applications. It currently supports the following Basic operations:

```
FOR ... TO ... (STEP)   GO TO              RUN        INT( )
ON ..... GOTO           IF ..... GO TO     LIST       TAB( )
ON ..... GOSUB          IF ..... THEN      NEW        ABS( )
PRINT                   LET                STOP       CHR$( )
INPUT                   REM                DIM        LOAD
READ                    DATA               GOSUB      SAVE
RESTORE                 NEXT               RETURN     LPRINT
POKE dec. addr, byte    PEEK(dec. addr)    DOS        LLIST
```

Decimal addresses in the above can be 0 through 16777215, but note that trying to PEEK or POKE a nonexistent address may lead to a BUS ERROR.

Although this Basic allows string constants (as in PRINT "HI"), it does not allow string variables (such as A$). Only one statement per line is allowed, the arithmetic is floating point BCD with nine significant digits, and no exponentiation is allowed. Variable names can be A through Z, and A0 through Z9, but array names can only be A() through Z(); one or two dimensions of 1 through 255 are allowed, but subscripts start with 1, not 0.

LPRINT and LLIST are similar to Print and List, respectively, but output to device 3 instead of the current output device (DEVOUT, usually 0).

This Basic is a modification of the (by now almost classical) 4K Basic written by Robert H. Uiterwyk for the first 6800 machines in the mid seventies. The Basic 6800 version of the interpreter is in the public domain and the original Version 2.1 source code is available in the book "Best of Interface Age, Volume 1: Software in BASIC", published in 1979 by dilithium Press, P.O. Box 92, Forest Grove OR 97116. We greatly recommend this book for further information about this interpreter.

Basic error messages are printed in the form

```
ERROR: ....... IN LINE xxxx
```

A description of the error replaces the periods, and the line number identifies the location of the error in the program. If no line number is given, the error occurred in a command or direct execution statement.

UNDELETE

UNDELETE is used to restore files which have been accidentally deleted. UNDELETE can only undelete one file at a time - the last file deleted - but it can be used again to undelete the file before that, and so on.

The syntax for using this command is

```
SK*DOS:  UNDELETE <drive-number.file-name.extension>
```

where the drive number and file name are required, but the extension defaults to .SCR if not given. If you do not know the correct name of the file, assign some dummy name, undelete the file, and then examine it to see what to rename it to. In any case, undeleted files should be examined to check that they are the desired file.

Depending on the size of the file and the size of the free space, the file may exist in the free space for a long time, or it may be written over quite soon. Hence files should be undeleted as soon as possible. In any case, once the free space of a disk is reorganized with REDOFREE, files can no longer be undeleted.

VERIFY

VERIFY is used to turn floppy disk verification on or off. The correct syntax is

```
SK*DOS: VERIFY ON
```

or

```
SK*DOS: VERIFY OFF
```

Normally, SK*DOS defaults to verifying all sectors it writes to a floppy disk. Disk operation, however, will be much faster if verification is turned off.

We feel that verification is desirable - after all, disks do occasionally make errors - and therefore SK*DOS normally has verification enabled unless you specifically turn it off. There are, however, some major manufacturers who feel that verification is not necessary, and whose systems default to verification off.

Although we do not recommend turning verification off, you may wish to do so for specific applications - for example, while using HTOF or while doing a backup. These are cases where you can easily check the accuracy of the new disk by doing a CHECKSUM on it.

For example, here is a way to do a very fast hard disk backup onto floppies by turning off verification:

(1) Use VERIFY OFF to turn off verification.

(2) Enable the disk CACHE, making sure it is large enough to contain the entire destination disk. For example, use a 750K cache for a 700K disk.

(3) FORMAT a fresh floppy disk once the empty cache is set up. This reads the entire empty disk into the cache.

(4) COPY files to the destination disk using the A option of COPY to copy in alphabetic order.

(5) Once the disk is full, use CHECKSUM on the destination disk. Since the data is still in the cache, you are actually checksumming the data in RAM, not the data on the disk.

(6) Use CACHE <drive-number> to flush the data from the cache, and CHECKSUM the disk again. The new checksum, which uses data actually read off the disk, must match the checksum in step 5.

(7) If there is more to backup from the hard disk, flush the cache again and go back to step 3. This time, however, give COPY appropriate arguments so it continues backing up files from where it stopped the previous time.

VERSION

VERSION is used to determine the version number of a binary file (if that file has a version number included). All SK*DOS utilities, and most other programs, begin with the sequence of instructions

```
START   BRA.S    START1
        DC.W     $0102
START1  ...      actual program begins here
```

so that the third and fourth bytes of the program are the version number (0102 in the above example, which is printed as 1.2). VERSION can be used to check that version code. To use it, type the command VERSION followed by the name of the file, as in

```
SK*DOS: VERSION COPY
```

If not entered, the file name defaults to a .COM extension.

XEQ

XEQ is a memory-resident command used to execute a program previously loaded by SK*DOS using the execution address loaded with the file from the disk (as modified by OFFSET).

If the program requires arguments as part of the calling line, these arguments can be placed after the XEQ. For example, suppose the command CAT 0 is used to display a catalog of a disk in drive 1; the command XEQ 1 would then repeat CAT, but would give it the argument 1 to specify drive 1.

Note that XEQ can only be used to repeat disk-resident programs, not memory-resident programs.

APPENDIX H. ADDENDA AND OTHER INFORMATION

My favorite law (which I affectionately call Stark's First Law of Computerdom) says "A program without bugs must be so simple that it probably wasn't worth writing in the first place."

It is quite likely that somewhere within SK*DOS lie one or more bugs which have yet to be discovered. If you find one, please let us know and we will exterminate it at once. In any case, be sure to follow the instructions on the User Registration page to be sure that we can contact you with updates if required.

This Appendix provides additional information about SK*DOS which does not seem to fit any other section, as well as addenda or corrections to the manual which are discovered after the remainder of the manual is printed.

1. When SK*DOS is first booted, it looks on the boot disk for a text file called STARTUP.BAT. This batch file may have a command which will be immediately executed as if it had come from the keyboard. For example, if this file contains the text

```
LIST READ-ME
```

then SK*DOS will list the contents of the file READ-ME.TXT after it asks for the date, just before it prints its first prompt.

2. To save memory space (as well as to allow some customization of messages), the one-line explanations printed with error messages are contained in a file called ERRCODES.SYS. Whenever PERROR prints out an error code, it looks on the current system drive for the ERRCODES.SYS file. If that file is present, then PERROR searches for the appropriate error number in that file and prints the text immediately following that number. You may LIST ERRCODES.SYS to see the file format, and may make changes with any editor to suit your taste.

The ERRCODES.SYS file contains error codes above 100, although SK*DOS itself only uses error codes through 30. These higher codes are used for 68K exception vectors. Each line of the file begins with a two-character error code number; in order to allow error codes through 240 to be represented with just two characters, the first digit of codes above 99 must be replaced by its corresponding ASCII character. For example, code 100 is represented by :0 since the colon comes after the 9 in the ASCII code, and thus the colon represents the digits 10.

APPENDIX I. ASM: THE 68000/68010 ASSEMBLER

by Edgar M. (Bud) Pass, Ph.D.

Copyright Notice

Limited Warranty Statement

Problems and Improvements

Users are encouraged to submit problems and to suggest or to provide improvements for the ASM Assembler. Such input will be processed on a best effort basis. Computer Systems Consultants reserves the right to make program corrections or improvements on an individual or wholesale basis, as required. The company is under no obligation to provide corrections or improvements to all users of ASM. In the case of specific situations requiring extensions to ASM or major assistance in its use, consulting is available on a pre-arranged, for-fee basis.

Introduction

The ASM assembler is used to convert 68000/68010 assembler language mnemonics and expressions to 68000/68010 machine language. The syntax of the assembler language which it accepts is compatible with the standard language as implemented in the Motorola 68010 Resident Structured Assembler.

It may also be used to produce machine language for any of the other processors in the 680xx family, although with some limitations. For those processors with instruction sets which are supersets of the 68010 instruction set, such as the 68020 and 68030, only those instructions common to the 68010 and to the superset processor may be generated directly by ASM. For those processors with instruction sets which are subsets of the 68010 instruction set, such as the 68000 and 68008, only those instructions common to the 68010 and to the subset processor may be correctly generated by ASM; those instructions not in the subset will be treated as invalid instructions by the subset processor. The M0 option may be specified on the command line to inhibit the incorrect generation of code for subset processors.

The ASM assembler provides an absolute assembler capability, and produces object code directly, without the use of a link-editor. It supports conditional assembly, the use of complex arithmetic expressions, and the inclusion of auxiliary source files. It produces an optional formatted listing of the program being compiled, including a sorted cross-reference of the symbols declared and used in the program, and generates English language error messages. A macro pre-processor is available with the full package.

Invocation of The 68000/68010 Assembler

The ASM assembler is invoked with a command line formatted as follows:

```
ASM program[.ext] [+/-options]
```

in which the following symbols are used:

program specifies the main program source file name; it must meet all requirements for a legal file name in SK*DOS.

ext represents an optional file extension; by default, ASM expects a source file name to have extension ".txt", if none is specified.

options represents optional assembler options, as described below; multiple options may be separated or may be joined together; the '+' or '-' symbol must be used to start each group of options; for example, the following string could represent a properly-formed set of options:

```
+blt4000d -M1
```

The following options are allowed:

+B, +b, -B, or -b

inhibits the construction of the output object file, which is produced by default; this file has the same file name as the main program source file, except that it has suffix ".mxt" (formatted as S-records) or ".com" or ".bin" (formatted as binary records), the details of which are described later.

+F, +f, -F, or -f

inhibits the generation of word-alignment bytes generated when DC.B directives are not followed immediately by DC.B directives.

+Hxxx, +hxxx, -Hxxx, or -hxxx

sets listing pagination mode and sets page length to decimal xxx, which may not be less than 24 or greater than 130.

+L, +l, -L, or -l

inhibits the generation of the formatted source listing; the source listing may be redirected to disk or printer with command line parameters, as supported by SK*DOS/68K; vertically formatted and paginated listings may be created with the H option or with auxiliary programs.

+Mx, +mx, -Mx, or -mx

indicates the type of processor for which code is to be generated: x is 0 for 68000, 1 for 68010, 2 for 68020, 3 for 68030 (2 and 3 are treated identically to 1, which is usually the default option).

+Ox, +ox, -Ox, or -ox

indicates type of object file to be produced: x is 0 for S3, 1 for S2, 2 for 02, 3 for 03, 5 for 05 (3 is the default option.)

+Pxxx, +pxxx, -Pxxx, or -pxxx

sets width of formatted output lines to decimal xxx; the default is 80, but it may be set to any value between 50 and 132; setting the output line width less than 64 will inhibit output formatting.

+S, +s, -S, or -s

inhibits the generation of the formatted and sorted symbol table listing produced at the end of the source listing on other than minimum systems.

+Txxxx, +txxxx, -Txxxx, or -txxx

sets symbol table size to decimal xxxx bytes, to attempt to force a smaller or larger symbol table size; since the maximum size of the symbol table in a given case depends upon the size of memory available to the assembler, if insufficient space is available to process a program, the number of program labels and symbols must be decreased, or the program must be segmented and separately assembled.

+X, +x, -X, or -x

inserts the formatted symbol table at the end of an S3/S2 object file.

If the assembler finds missing or invalid information specified on the command line, it will output a prompt message indicating the correct format of the command line. On other than minimum systems, this prompt provides summary information on the valid command-line arguments (file names and options).

Assembler Syntax

The input to this assembler is one text file containing a number of lines formatted according to the requirements of the 68000/68010 assembler language. Because of the ability of this assembler to include auxiliary text files, source text libraries may be established and used. Whatever the source, the output lines are automatically formatted to separate the label, instruction, and operand fields, whenever the output line is at least 64 characters in width.

The statements acceptable to this assembler are free-field, one statement per line. Intermediate fields are separated by at least one space, and each line is terminated by a carriage return.

There are three types of statements in the 68000/68010 assembler language, as follows:

instruction
directive
comment

Each of these statement types is discussed below.

Instruction Statement

An instruction statement consists of zero to four fields, as follows:

[label[:]] [instruction [operand] [comment]]

Label Field

The label field is optional. It is normally used to provide a symbolic name for the address of the code generated by the following assembler instruction. If the label field is omitted, it must be replaced with a space.

Labels are composed of one to 30 characters, of which all characters are significant. The first character of a label must be a letter or period, and the remaining characters must be letters, periods, digits, underlines, and dollar signs. Lower case and upper case letters are considered distinct. Label definitions starting at other than the left margin must end with a colon.

The following symbols may not be used as labels, since to do so would create ambiguous register references:

A0 thru A7, CCR, D0 thru D7, DFC, PC, SFC, SP, SR, USP, VBR,
a0 thru a7, ccr, d0 thru d7, dfc, pc, sfc, sp, sr, usp, vbr.

These represent the following 68000 or 68010 registers:

A0 thru A7 32-bit address registers A0 thru A7,
CCR 8-bit condition code register (low 8 bits of SR),
D0 thru D7 32-bit data registers D0 thru D7,
DFC Destination function code register,
PC 32-bit program counter,
SFC Source function code register,
SP 32-bit stack pointer (same as A7),
SR 16-bit status register,
USP 32-bit user stack pointer,
VBR Vector base register.

Instruction Field

The instruction field contains the 68000/68010 assembler instruction. It will normally always be present, but if it is omitted, the operand and comment fields must also be deleted. Lines containing labels only are allowed, as are totally-blank lines.

A 68000/68010 instruction name contains no more than seven characters, with upper case and lower case non-unique. The complete set of 68010 assembler language instructions is described in the Motorola 68010 Resident Structured Assembler manual.

Those 68000/68010 instructions which require data size specification may be suffixed with length modifiers. These include ".B" for 8-bits, ".W" for 16-bits, and ".L" for 32-bits. If no data length modifier is specified, ".W" is normally assumed.

To indicate short branch instructions, the suffix ".S" may be used. This assembler will not change short branches to long branches or vice versa, and a short branch to the next instruction will not be changed to a long branch, but will be noted as an illegal operation on the 68000/68010.

Operand Field

The operand field contains zero or more parameters of the instruction. If multiple sub-fields are present, they must be separated by a comma. Only the MOVEM instruction allows more than two operands, but the register list of that instruction must be separated with slashes and hyphens, even if only one register is used. Many instructions (such as MOVE) have two operands, which are designated as source and destination sub-fields, from left to right.

This assembler supports the standard Motorola notation for the 68000/68010 effective addressing modes. They are summarized below.

CCR, DFC, SFC, SR, USP, VBR - Control register

An - Address register direct

Dn - Data register direct

(An) - Address register indirect

(An)+ - Address register indirect with postincrement

-(An) Address register indirect with predecrement

expression(An) - Address register indirect with displacement

expression(An,Xn); expression(An,Xn.W); expression(An,Xn.L) - Address register indirect with index

expression (in ORG); *-expression; *[+expression] (in ORG) - Address short or long

expression (in RORG); *[+expression] (in RORG); expression(PC) - Program counter with displacement

expression(PC,Xn); expression(PC,Xn.W); expression(PC,Xn.L) - Program counter with displacement and index

#expression - Immediate

Comment Field

The comment field consists of any text following the fields described above, but preceding the terminating carriage return on the line. It may contain characters with hex values from $20 thru $7E.

Directive Statement

A directive statement consists of one to four fields, as follows:

[label[:]] directive [operand] [comment]

Label Field

The label field of a directive follows the same rules provided above. However, a label may be used only with the directives indicated below.

Directive Field

The directive field provides information for the assembler, rather than instructions for the 68K processor. This information includes such items as the base address of the program, the establishment of symbol values, the allocation of storage, the specification of additional source files to be read by the assembler, etc. A list of the directives implemented by ASM is provided later in this manual. Lower case and upper case letters in the directive field are not considered distinct.

Operand Field

The operand field consists of zero or more sub-fields. Multiple sub-fields are separated with commas.

Comment Field

The comment field consists of any text following the fields described above, and preceding the terminating carriage return. The comment field may be composed of characters with ASCII hex values from $20 thru $7E.

Comment Statement

A line which starts with an asterisk is ignored by the assembler except for being optionally listed (along with the remainder of the program). A comment statement may be composed of characters with ASCII hex values from $20 thru $7E.

Directives

```
[label[:]] DC    operand[,operand[,...]]
[label[:]] DC.B  operand[,operand[,...]]
[label[:]] DC.L  operand[,operand[,...]]
[label[:]] DC.W  operand[,operand[,...]]
[label[:]] FCB   operand[,operand[,...]]
[label[:]] FCC   operand[,operand[,...]]
[label[:]] FDB   operand[,operand[,...]]
```

DC specifies the assignment of one or more values into successive words, bytes, or long words; the suffixes are interpreted as described above for instruction suffixes; a DC.B directive not followed immediately by another DC.B directive will normally have a filler byte inserted, if necessary, to force word alignment (except when the +/-F option is used.) FCB and FCC are equivalent to DC.B and FDB is equivalent to DC.W. Strings must be enclosed in single or double quote symbols.

```
[label[:]] DS    expression
[label[:]] DS.B  expression
[label[:]] DS.L  expression
[label[:]] DS.W  expression
[label[:]] RMB   expression
```

DS indicates the reservation of a specified number of words, bytes, or long words, without placing values into the locations; no word alignment is forced after any DS directive, although word alignment is forced before DS and DS.W directives, and double-word alignment is forced before DS.L directives. RMB is equivalent to DS.B.

```
END [label]
```

END indicates the logical end of the assembler language unit, although not necessarily the end of the program, as the assembler will continue to process input until it finds physical end of file. Although the END statement may not be

labelled, an optional label may appear as an operand indicating the starting address of the program, and will be placed into the object file as the transfer address.

```
ENDC
ENDIF
```

ENDC and ENDIF indicate the end of the range of an IFxx or IFDEF declarative.

```
label [:] EQU expression
```

EQU defines a symbol and sets its value to that of the single operand; this operand may contain complex expressions, but not forward references, and once defined by an EQU statement or as a program label, a symbol's value may not be changed.

```
IFxx expression
IFDEF label
IFP1
```

IFxx or IFDEF indicates the beginning of a sequence of assembler statements which may be logically conditionally included or excluded from the input to the assembler; the xx indicated above must be replaced by EQ, GE, GT, LE, LT, or NE, in order to indicate the condition (expression xx zero) for the inclusion of the assembler statements, which are terminated by a corresponding ENDC/ENDIF directive; IFDEF includes the sequence of statements if the indicated label is defined; IFP1 includes the sequence on the first pass of the assembler, but excludes it on the second pass; IFxx, IFDEF, and IFP1 statements may be nested to 32 aggregate levels.

```
LIB filename
USE filename
```

LIB and USE provide the name of an auxiliary source file which is logically inserted into the current source file in place of the LIB or USE directive; the file name must follow the conventions required by SK*DOS/68K.

```
LIST
```

LIST allows the source listing to be produced, unless it is suppressed by an option on the command line.

```
NOLIST
```

NOLIST suppresses source listing production.

```
NOPAGE
```

NOPAGE suppresses source listing pagination.

```
ORG expression
```

ORG specifies the starting absolute memory base of the program counter, assuming short absolute addresses restricted to the first and last 32768 bytes of address space.

```
ORG.L expression
```

ORG.L specifies the starting absolute memory base of the program counter, assuming long absolute addresses representing the full range of the address space.

```
PAGE
```

PAGE causes the next line on the source listing to appear on the next page.

```
RORG expression
```

RORG specifies the starting absolute memory base of the program counter, assuming short absolute addresses, and the generation of program-counter-relative effective addresses.

```
RORG.L expression
```

RORG.L specifies the starting absolute memory base of the program counter, assuming long absolute addresses, and the generation of program-counter-relative effective addresses.

```
RPT expression
```

RPT specifies the number of times the next line in the same source file will be repeated; the next line is always included at least once, even if the value of the expression is zero.

```
label [:] SET expression
label [:] = expression
```

SET defines or redefines a symbol and sets its value to that of the single operand; this operand may contain complex expressions, but not complex forward references, and this symbol's value may be changed only by another SET or = directive.

```
SPC
```

SPC places a blank line in the source listing.

```
TTL title
NAM title
```

TTL and NAM specify the title to be placed at the top of each page in the source listing when pagination is active.

Arithmetic Expressions

Expressions consist of combinations of symbols, constants, operators, and parentheses. Symbols must normally be defined before being used in complex arithmetic expression evaluations. All arithmetic and logical operations are performed using 32-bit two's complement integer arithmetic; division is truncated, and overflows are normally lost. Expressions may be forced into 16-bit mode by enclosing them with '(' ... ').W', or by masking them with $ffff.

Symbols used in expressions include the following:

program labels
symbols defined with DC, DS, EQU, SET directives
numeric constants
'*' (the location counter)

Program labels and symbols are composed of one to 30 characters, starting with a letter or period, as described earlier in this manual.

Constants may be expressed in decimal, hexadecimal, binary, octal, or ASCII, and are constructed as follows:

Decimal constants are represented by the digits 0 thru 9; they may not contain decimal points.

Hexadecimal constants start with the dollar symbol, which must be followed by one or more digits and/or letters, restricted to the ranges a thru f and A thru F.

Binary constants start with the percent symbol, which must be followed by one or more 0's and 1's.

Octal constants start with the at symbol, which must be followed by one or more digits, restricted to 0 to 7.

ASCII constants start and end with single or double quote symbols, and contain characters with hex values $20 thru $7E; quote symbols within the constants, matching the delimiters, are represented by two adjacent quote symbols; the number of characters in ASCII constants within expressions is restricted to four except in DC and equivalent types of directives.

Operators used in expressions may include the following, in decreasing order of priority:

unary minus -
left/right logical shifts < < and > >
logical and/or & and !
division/multiplication / and *
addition/subtraction + and -

Object File Formats

One of the object file types produced by ASM assembler follows the format of the standard Motorola S-records file. The format of each type of S-record is described below.

The record type field is an 'S' and a digit, interpreted as follows:

S1 indicates a record containing data, starting at the value in the 16 bit (2 byte) address field.

S2 indicates a record containing data, starting at the value in the 24 bit (3 byte) address field.

S3 indicates a record containing data, starting at the value in the 32 bit (4 byte) address field.

S7 indicates a record containing an optional 32 bit (4 byte) transfer address, and terminates a group of S3 records.

S8 indicates a record containing an optional 24 bit (3 byte) transfer address, and terminates a group of S2 records.

S9 indicates a record containing an optional 16 bit (2 byte) transfer address, and terminates a group of S1 records.

The number of bytes to follow is indicated as two hexadecimal digits. If an address or value is to appear in this record, it follows the length field just described; it is represented in hexadecimal. If data appears in the record, it follows the starting address, and is represented in hexadecimal. The one's-complement checksum of all of the items is indicated as two trailing hexadecimal digits.

Another set of the object file types produced by ASM is the SK*DOS/68K binary load records 02, 03, and 05. These formats are described below:

leadin	address	length	body	comments
$02	2-byte-offset	1-byte	data	
$03	4-byte-offset	2-byte	data	default
$05	4-byte	2-byte	data	
$16	2-byte-offset	none	none	transfer address
$18	4-byte-offset	none	none	transfer address, default
$19	4-byte	none	none	transfer address

The naming of the object file produced by ASM is dependent upon the type of object file and the system on which it is run. The following table provides the extensions:

type	extension
02/03/05	.com
S3/S2	.mxt

Instruction Syntax

The syntax of the assembler language accepted by this assembler is compatible with the standard language as implemented in the Motorola 68010 Resident Structured Assembler.

Upper and lower case letters in instructions are not considered distinct. For those instructions with length modifiers, the suffix ".W" will usually be assumed if none is provided in the instruction field. Branch instructions must be coded with ".S" to indicate short branches or with ".L" or without suffix to indicate long branch instructions. A short branch to the next instruction is invalid on the 68010 and will not be changed to a long branch. For unsized instructions, length modifiers are generally ignored.

Certain instructions (ADD, AND, CMP, EOR, MOVE, NEG, OR, and SUB) have variations in their basic operation codes beyond length modification. The following variations are normally handled by this assembler, but may be coded explicitly, as follows:

....A address register operation
....I immediate operation

Other variations must be coded explicitly (as the assembler has no manner in which to determine that they were desired), as follows:

....M memory operation
....Q quick immediate operation
....X extended carry operation

For the MOVEM instruction, if only one register is present in the register list, it must appear as Dx-Dx, Ax-Ax, Dx/Dx, or Ax/Ax. The syntax of the MOVEM requires a register list, not a register.

The BKPT, MOVEC, MOVEM, and RTD instructions and the MOVE CCR,... and MOVE ...,CCR variants of the MOVE instruction are invalid when the 68000 option (M0) is chosen on the command line, as these are all 68010 instructions.

Error Messages

Most error messages are self-explanatory; however, all of them are briefly explained in the list below. For minimum systems, only error numbers are output by the assembler.

19 Assembler error - An internal error in the assembler has been detected, probably caused by syntax errors in the statement.

07 Bad 16 bit displacement - The displacement value is less than -32768 or greater than +32767.

25 Bad 16 bit displacement range - The displacement value is less than -32768 or greater than +32767.

11 Bad 16 bit extension - The extension value is less than -32768 or greater than +32767.

02 Bad 3 bit value - The quick immediate data value is not in the range 1 to 8, or the specified breakpoint number is invalid.

03 Bad 32 bit field specifier - The bit field value specifies a bit outside of a long word.

31 Bad 4 bit value - The specified vector number is invalid.

08 Bad 8 bit displacement - The displacement value is less than -128 or greater than +127.

26 Bad 8 bit displacement range - The displacement value is less than -128 or greater than +255.

12 Bad 8 bit extension - The extension value is less than -128 or greater than +255.

15 Bad 8 bit field specifier - The bit field value specifies a bit outside of a byte.

22 Bad 8 bit operand - The operand value is less than -128 or greater than +255.

01 Bad address displacement - The displacement is out of range for the effective address type.

06 Bad count operand - The immediate shift count is less than 1 or greater than 8.

10 Bad destination effective address - The destination effective address type is invalid for this type of instruction.

09 Bad destination indexed address - The indexed destination effective address type is invalid for this type of instruction.

13 Bad effective address - The effective address type is invalid for this type of instruction.

14 Bad expression format - The expression is badly-formed or contains a space.

27 Bad expression range - The address expression has a value outside the short addressing range.

18 Bad instruction field - The item in the instruction field cannot be recognized.

20 Bad label usage - The directive statement may not be labelled.

xx Bad library file ... - The indicated library file could not be found.

21 Bad multiple destination - The operand specifies registers not allowed in a multiple destination list.

xx Bad object file ... - The indicated object file could not be created.

16 Bad operand 1 for instruction type - The first operand is improper for this instruction.

04 Bad operand 1 format - The first operand has been coded incorrectly.

17 Bad operand 2 for instruction type - The second operand is improper for this instruction.

05 Bad operand 2 format - The second operand has been coded incorrectly.

23 Bad operand format - The instruction operand field is badly-formed or contains an unintended space or other illegal character.

xx Bad source file ... - The indicated source file could not be found.

xx Library file nest error ... - The indicated library file could not be included, as the nest level is too deep (usually 3 for minimum systems and 12 for others).

24 Phasing error - The non-redefinable symbol has been found to have a different value on the second pass than it had on the first pass of the assembler; this is often due to conditional assembly including different text on the two passes or due to other syntax errors in the assembly text.

28 Redefined symbol - The symbol defined in other than a SET statement has been redefined.

30 Symbol table overflow - The table which contains the program labels has been filled to capacity or has been specified as too large; attempt to change its size with the command line option, reduce the number of labels used in the program, reduce their lengths, or divide the program into smaller parts.

29 Undefined symbol - The symbol is used in an expression or in a context which does not allow forward references or is not defined in the program.