SK*DOS 68K
Configuration Manual

by Peter A. Stark

All Star-K computer programs are licensed on an "as is" basis without warranty.

Star-K Software Systems Corporation shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Star-K, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

SOFTWARE LICENSE

A. Star-K Software Systems Corp. grants to customer a non-exclusive, paid up license to use on customer's computer the Star-K computer software received. Title to the media on which the software is recorded (cassette and/or disk) or stored (ROM) is transferred to customer, but not title to the software.

B. In consideration of this license, customer shall not reproduce copies of Star-K software except to reproduce the number of copies required for use on customer's computer and shall include the copyright notice on all copies of software reproduced in whole or in part.

C. Although the source code for HUMBUG and parts of SK*DOS are supplied with this manual, such code is provided strictly for the convenience of the purchaser in implementing SK*DOS on his system, and is not to be used for any other purpose.

D. The provisions of this software license (paragraphs A through D) shall also be applicable to third parties purchasing such software from customer.

NOTES

SK*DOS and HUMBUG are registered trademarks of Star-K Software Systems Corp. Whenever used in this manual, the term FLEX is a trademark of Technical Systems Consultants Inc.

6809 SK*DOS was formerly known as STAR-DOS.

We provide support for SK*DOS users via the Star-K Software BBS at 914-241-3307. This computerized system operates 24 hours a day at 300 and 1200 baud, 8 bits, no parity, 1 stop bit. Feel free to call the BBS at any time - it is a popular medium for interchanging ideas, opinions, and programs among the many users of Star-K software and hardware.

This is revision 1.09 of the manual, last revised on May 25, 1991.

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 at all possible, you should make backups of your original SK*DOS system disk before proceeding. You may do so on a system running SK*DOS (either 6809 or 68K) or FLEX. It may also be possible to make a backup on other computers if they allow a "mirror image" copy of one disk to another.

SK*DOS system disks are supplied in several different formats. Disks configured for a specific system are generally supplied in the format most suitable for that system. For example, 68K SK*DOS boot disks are generally double-density, double-sided, 5-1/4", with 40 tracks, numbered 0 through 39, and with 18 sectors per track.

The disks supplied with this manual, on the other hand, may be different, depending on the system they are intended for. For example, those disks intended for use with a 6809 Flex or SK*DOS system will be 5-1/4", single density, with 40 tracks numbered 0-39, and 10 sectors per track, numbered 1-10. They may also be supplied as 8", single density, with 77 tracks numbered 0-76, and 15 sectors per track, numbered 1-15. Disks intended for reading on an IBM or compatible PC will be double-density, double-sided.

During the entire configuration process, make sure to write-protect your SK*DOS disks at all times. On a 5-1/4" disk this is done by putting an opaque tape strip over the notch on the side of the disk; on an 8" disk it is done by making sure there is no tape on the write-protect notch at the rear of the disk.

CHAPTER 1 - INTRODUCTION

This manual describes how to adapt the SK*DOS/68K Disk Operating System to systems different from those normally supported. In order to do that properly, it is important to fully understand how disk operating systems work and what they do. The next chapter will give you a short overview of SK*DOS, but before proceeding further, you should read the SK*DOS/68K User's Manual. (This manual does not repeat any of the information in the User's manual.) In addition to this manual, your SK*DOS Configuration Pack includes one or more disks which contain source code for the various routines you will have to write or adapt to your system. These disks are essential to your configuration, so treat them gently.

What Systems Star-K Supports

Although most 68K systems are similar in many ways, they sometimes have major differences in their internal or I/O structure, and in how they are programmed. For that reason, it is difficult for us to support every conceivable kind of system. We therefore provide off-the-shelf versions of SK*DOS for a small number of the more popular 68K systems, and rely on our customers to do their own adaptation to more unusual situations.

In addition, some manufacturers have provided interface software for adapting SK*DOS to their equipment, or have done the adaptation themselves. It is thus entirely possible that the copy of SK*DOS you have obtained is already configured to run on the specific equipment you have. In that case, all you need do is to boot the system as described in the User's Manual. (And you should try to do so ... but make sure that your master SK*DOS disk has a write-protect tape installed!) You may not even need this Configuration Manual.

If, however, you have hardware for which a specific version of SK*DOS does not yet exist, then this manual will provide information to allow you to adapt it to your system.

BUT NOTE: Such adaptation requires a good knowledge of machine language programming and disk system theory. It may be better for you to contact the manufacturer of your equipment, and ask them for assistance. It is quite likely that they already have similar software developed for use with another DOS, and that they can supply the adaptation to SK*DOS with little effort on their part.

The Files on the Configuration Disk(s)

Depending on the disk format, there may be one or more Configuration Disks supplied with this manual. The following files are on these disks:

SK*DOS.S19	SK*DOS in S1-S9 format, less BIOS, for loading via a serial line
SK*DOS.COR	Binary code for SK*DOS, less BIOS
HUMBUG.TXT	Source code for a debugging system ROM
BOOTROM.TXT	Source code for a boot ROM
BIOS.TXT	Source code for the BIOS
FORMAT.TXT	Source code for the floppy disk FORMAT program
HDFORMAT.TXT	Source code for the hard disk FORMAT program
SKEQUATE.TXT	Library file
P.TXT	Source code for the printer driver program
DATEADD.TXT	Source code for the calendar date routine
TIMEADD.TXT	Source code for the time stamping routine
TIME.TXT	Source code for the clock set utility
PARK.TXT	Source code for a hard disk park utility
SEND.TXT	Source code for a S1/S9-to-keyboard translator
SENDPROM.TXT	Source code for a S1/S9-to-Intel format translator

What You Need to Configure SK*DOS for Your Equipment

To configure SK*DOS to run on a new computer, you essentially need four things:

1. The new 68K computer itself; in the rest of this manual, this is called the 'target' computer. Information on the minimum target computer configuration appears below.

2. Some way to edit and assemble the system- specific software which has to be adapted to your target system. Although some of these - such as FORMAT - can conceivably be done on the target 68K system itself, the others - boot routines, disk and console drivers - cannot, since they must be properly adapted before SK*DOS itself will even run. Unlike earlier 6809 systems, where it was possible to hand-assemble code, this is not a practical alternative for 68K systems due to the extreme complexity of the machine language. Hence you will need some other system, called the 'host' system, on which to do this conversion. Moreover, you will need a text editor and 68000 cross-assembler which run on that host system. (But note: if your target system is already configured to run some other DOS, then the target system can also act as the host.)

3. Some way to get assembled object code from the host system into the target system. This is best done by transferring object code from the host system into the target system on a serial RS-232C link. Alternatively, if the host system is another SK*DOS system (either 6809 or 68K) or Flex (6809) system, then its disk format is compatible with the 68K SK*DOS disk format. Then object code can be written to a disk and read in with the aid of a simple disk-read routine which can be entered by hand into the 68K target system.

4. Finally, some sort of a ROM is needed on the 68K target system which (a) provides a way of initializing the system, (b) provides facilities for debugging 68K object code, and (c) provides a way of accepting serial data from an RS-232C line and storing it in memory, if that is how you are transferring object code into your target system. At the very least, that ROM must be able to enter machine code into memory, and start and stop programs. The more functions that program has, the better; ideally, it should also contain simple character input and output

routines which can be used by SK*DOS as well. A disk read routine which can read a disk sector into memory would also be very useful.

What the Target System Needs

Your 68K target system needs to have the following:

1. A minimum of 32K of RAM. Of this, about 24K will be used to hold SK*DOS, while the remainder is needed for the minimum utilities. But to be really useful, your system should have more memory - a lot more. To run application programs, you may need 128K or more (for example the stock SK*DOS assembler requires more than 128K of memory). In addition, you may want additional memory for a RAM disk.

2. A boot ROM which will bring up the basic system and then load SK*DOS from disk. A general purpose ROM monitor may be useful, but even a special purpose 'boot-only' ROM will do the job. This topic is discussed further in the next two chapters.

3. At least one, and hopefully two or more, disk drives with an appropriate disk controller. Although any kind of a disk drive can be used to run SK*DOS (with the appropriate driver programs), you will need a 5-1/4" floppy disk drive for the initial work of reading the SK*DOS disk we supply, and implementing SK*DOS on your system.

4. A means of communicating with this computer, such as a CRT terminal. This device will probably already be able to communicate via your monitor ROM.

What Kind of Host System is Suitable?

Although many kinds of host systems will do the job, there are two that are most suitable - either a standard 6809 system (such as an SS-50 bus system, or a single-board computer such as the PT-69 from Peripheral Technology), or a Radio Shack Color Computer. In either case, the system should run either Flex, or preferably SK*DOS. SK*DOS for SS-50 and single-board 6809 systems is available from us; SK*DOS for the Color Computer is available from Computer Publishing Inc., 5900 Cassandra Smith, Hixson TN 37343 (615-842-4600, or Telex 5106006630). In general, an SS-50 or single-board system is preferable; the PT-69 is a particularly good buy if you do not yet have such a host system, as an assembled and tested board currently lists for $279; all you need is a power supply (easy to get), one or two disk drives and a terminal (which you probably already have for use with your 68K system) and the SK*DOS. The board and a matching SK*DOS are available from Peripheral Technology, 1480 Terrell Mill Rd., Marietta GA 30067 (404-984-0742).

To go along with the 6809 system, you will need an editor and a 68000 cross-assembler. (Any 6809 Flex or SK*DOS software you purchase for your host system is also usable on your 68K system using the SK*DOS09 6809 simulator.) Some additional software which will make your configuration easier is supplied on your SK*DOS disk. A second-best host system is an IBM or compatible PC or XT computer, since we can also supply SK*DOS configuration source code on an MS-DOS configured disk. As before, you will need an editor (or word processing program) to prepare source code, and a 68000 cross-assembler to assemble it; we can supply a 68000 cross-assembler for $50. In addition, you will need some communications software to allow you to send object code out the serial port to your target system.

How is SK*DOS Configuration Generally Done?

Configuring SK*DOS to run on a new system can be simple or it can be complicated - it depends on many factors. In fact, the exact procedure depends on just how close that system is to existing systems on which SK*DOS already runs, what kind of host system (if any) you have, what kind of supporting hardware and software is available to you, and how knowledgeable (and/or foolhardy) you are. Since there are so many variables, we won't give you a specific procedure to follow. But most users will perform the configuration in roughly the following order:

a. If your target system does not have an adequate ROM (or if it has no ROM, or if its ROM is designed for some other purpose and is not suitable for booting SK*DOS), then you will have to prepare such a ROM for your system. Chapter 3 describes how to do this.

b. Next, you must write the BIOS, or Basic Input/Output System. This consists of console drivers so that SK*DOS can communicate with you via the keyboard and display, and disk drivers so SK*DOS can read and write on disks. This is described in Chapter 4.

c. Next, you must transfer both the SK*DOS and the BIOS into your target system, preferably through the serial link. If all goes well, this will give you a running SK*DOS system.

d. Next, you must write a FORMAT program so you can format new disks and write on them. If everything has worked so far, you can do this using an editor and 68K assembler running under SK*DOS on your target system. This is described in Chapters 5 and 6.

e. Finally, you may add some of the frills described in Chapters 7 through 9.

Things become much more complicated if you don't have a host system. In this case, it is essential that your target system have a fairly good ROM monitor program which has at least a single-line assembler and disassembler. To assemble your required programs, you will have to combine the ROM's assembling abilities with your own pencil-and-paper record keeping to go through the following steps:

a. A sector read routine must be written to allow you to read a single sector from a disk. You will need this routine to allow you to read parts of the supplied SK*DOS system disk. The sector read routine is then incorporated into a "super-boot" program which will allow you to load SK*DOS into memory from the system disk. This allows you to load SK*DOS into memory, but does not yet give you a runable version.

b. Next, you must write the BIOS, or Basic Input/Output System. This consists of console drivers so that SK*DOS can communicate with you via the keyboard and display, and disk drivers so SK*DOS can read and write on disks.

c. Next, you must load SK*DOS with the super-boot, and combine it with your BIOS to obtain a working copy of SK*DOS in memory.

d. Next, you must write a FORMAT program so you can format new disks and write on them. If everything has worked so far, you can do this using an editor and 68K assembler running under SK*DOS on your target system.

Although the ultimate aim may be to provide a lot of bells and whistles, the main idea of the above procedure is to start

small and simple. Once you have SK*DOS up and running in a simple way, with just single-density and one-sided disks, then you can elaborate and extend it to double density, two sides, or even hard disks. But that time does not come until later.

Once SK*DOS is up and running, you will be able to use its editor and assembler for future revisions to the system. From time to time, you may be supplied with update versions of the DOS; you will then reassemble your BIOS on your 68000 system and append the BIOS object code to the furnished SK*DOS object code to form a loadable SK*DOS module.

We will return to the configuration procedure in Chapter 3, after we look at some more specific details about SK*DOS, in Chapter 2.

CHAPTER 2 - A SK*DOS OVERVIEW

This section gives an overview of SK*DOS and some of its more important details. Before reading it, however, you should be familiar with the contents of the SK*DOS/68K User's Manual, since this manual will not repeat any of the information in that manual except if it is specifically needed to understand some specific point.

What is a DOS?

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. As a secondary function, the DOS usually also contains various subroutines, such as character input or output routines, for communicating with other I/O devices in a simple and consistent manner.

What is SK*DOS?

SK*DOS consists of four 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 subroutines 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.

(4) The Basic Input-Output System or BIOS, which is the interface between SK*DOS and the hardware of your system. This manual primarily describes this BIOS and how to write one.

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 1000 tracks with 64 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 spindle. 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 tracks (numbered 0 through 255) per drive and 256 sectors (numbered 0 through 255) per track (but note that floppy disks, by convention, have sectors numbered beginning with 1 and thus there is no sector 0 on any track of a floppy disk.) The exact positioning of those sectors and tracks is controlled by the BIOS and FORMAT programs, not by SK*DOS itself.

[A short detour at this point, included more for completeness than because of an immediate need. We often differentiate between a 'logical address' and a 'physical address'. The physical address is the actual address where the hardware looks to find a particular item; the logical address is the address where the software thinks that item is located. In most systems the logical and physical addresses are the same, but it is possible to make them different. In that case there has to be some sort of conversion table or map which converts the logical address into a physical address so the desired item can be found quickly. SK*DOS refers to locations on a disk by their track and sector numbers; this is the SK*DOS logical address. Usually the data will in fact be on that physical track and sector. But it is possible for the disk drivers to convert each logical track and sector number into a (different) physical track and sector number where the data is really located, and go to that track and sector instead. This is often done with Winchester hard disks. End of Detour.]

Each sector in turn 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 thus has a present limit of about 16 megabytes of data per logical drive, computed as follows: 256 tracks x 256 sectors x 252 bytes/sector = 16,515,072 bytes Although this is a limit on the size of a logical drive, it does not limit the size of a physical drive, since a larger physical drive could be treated as two or more logical drives. Since SK*DOS allows up to ten logical drives, SK*DOS's maximum present on-line storage capacity is about 165 million bytes. Moreover, since drives can be reassigned with the DRIVE command, additional physical drives can be connected as well to provide a greater capacity.

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 0 and 1 - pointer to next sector
Bytes 2 and 3 - sector counter
Bytes 4 through 255 - 252 bytes of data

Some sectors have a slightly different format, and may omit the pointer and/or sector counter. All the tracks on a disk can be used for storing data and program files except for track 0. Track 0 is special in that the sectors on this track have special uses as follows:

Winchester Data Table (WINTAB)

Sector 0 of track 0 of a hard drive contains data describing the partitioning of the disk and other physical parameters. This will be described later.

The Super-Boot

Sector 1 on track 0 of a floppy disk holds the super-boot program. This is a program which is loaded by the boot program (which is usually located in the boot ROM) and which in turn loads the rest of SK*DOS into memory. (This sector has a full 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.) Sector 2 is often used as an extension of sector 1 if the super-boot is too long to fit into the first sector.

The System Information Sector (SIS)

Sector 3 is the System Information Sector or SIS. It contains the following information:

Bytes 16-26	Disk name (and extension)
Bytes 27-28	Disk number
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

All other unused bytes of the SIS are 00.

Sector 4 is generally empty, but is used by the COPY program to test whether the disk is write protected.

The Directory

The directory begins on sector 5 of track 0 and fills up the rest of the track. The first 16 bytes of each directory sector are empty, and the remaining 240 bytes hold directory entries. Since each directory entry requires 24 bytes, there is room for 10 entries in each sector. For example, on a 5-1/4" double-density single sided disk, there are 18 sectors in track 0. Hence there are 14 sectors in the directory, numbered from 5 to 18, for a total of 140 directory entries. The 14 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.)

Note several items: if the disk is double-sided, then the directory will also continue on track 0 of side B. The super-boot must be capable of reading both sides of a disk to properly boot SK*DOS. In some cases, disks may also be partially or entirely single-density (if they have been prepared for use with a 6809 SK*DOS or Flex system).

The 24 bytes in each directory entry are used as follows:

Bytes 0-10	File name (8 bytes) and extension (3 bytes)
Byte 11	File protection status (see PROTECT command)
Byte 12	Reserved for future use
Bytes 13-14	Track and sector number of first sector
Bytes 15-16	Track and sector number of last sector
Bytes 17-18	Number of sectors
Byte 19	Random file indicator (non-zero = random)
Byte 20	Time or sequence number
Bytes 21-23	Month, day, and year of file creation

The 24 bytes of each directory entry match bytes 4 through 30 of the corresponding FCB (except for a few empty bytes in an FCB) and are explained further in the User's Manual (see especially Chapter 13 of the User's Manual for a discussion of subdirectories.)

Note also that the track and sector numbers listed in both the directory as well as the SIS are logical track and sector numbers, and may not necessarily be the same as the physical track and sector numbers, although in most instances they will be. When a disk is initially formatted, the entire directory is filled with zeroes. Whenever SK*DOS encounters a directory entry which begins with 0, it assumes that the remainder of the directory is still empty and has never been used, and thus stops reading. When a directory entry is deleted, however, the first character of the file name is replaced by $FF. When SK*DOS encounters a directory entry beginning with $FF, it skips that entry and goes to the next.

SK*DOS Memory Map

Depending on where RAM is located in your 68K system, SK*DOS may have to be positioned at different addresses. In 99.9% of the cases, however, 68K systems will have RAM beginning at address 0 and extending upward in a contiguous block; in that case, SK*DOS will begin at location $1000. The rest of this manual will assume that this is the case. If your system does not fall into this category, contact us for a custom version of SK*DOS which is ORG'ed at a different address.

The system memory map will then look like the following:

$0000 to 00BF	Exception vectors
$00C0 to 09FF	Reserved for use by the ROM monitor, or by other system-specific programs
$0A00 to 0BFF	Reserved for booting
$0C00 to 0DFF	User stack
$0E00 to 0FFF	SK*DOS stack
$1000 to 11FF	SK*DOS-to-BIOS vectors
$1200 to OFFINI	SK*DOS program, including the BIOS you supply
OFFINI to OFFSET	Temporary programs which may be called by the SK*DOS user
OFFSET to MEMEND	Available user memory
MEMEND to end of memory	temporary programs which may be called by the SK*DOS user
Somewhere below $FFFFC000	I/O equipment
Very top of memory space	ROM system monitor such as HUMBUG or a boot ROM.

Initially, first versions of SK*DOS will have OFFINI set to approximately $6000 to $8000; later versions will use a higher address as the DOS itself, as well as later versions of the BIOS, will become larger.

As the above table shows, the BIOS does not have a specific address - the only requirement is that it lie between the top of SK*DOS and the value of OFFINI. But both of these limits are actually moving targets.

As supplied on the SK*DOS distribution disk, SK*DOS.COR contains the core of SK*DOS, but without any BIOS at all. Since it has no BIOS, OFFINI points not to the top of the BIOS, but to the top of SK*DOS itself. This tells you where to ORG the BIOS when you write it. Once the BIOS is appended to SK*DOS.COR, then it is up to you to change OFFINI so it points above the BIOS, rather than just to the top of SK*DOS.COR. In that context, the initial OFFINI will change

whenever we send you an updated version of SK*DOS, so that you may have to set a new ORG on your BIOS and reassemble it to fit the latest SK*DOS version (unless you left some space between the top of SK*DOS and the beginning of SK*DOS in your prior version.) Likewise, the new value of OFFINI will change each time you reassemble your BIOS or make any other changes to it. Although this seems to complicate the entire picture, it has the advantage of leaving almost unlimited room for expansion of both SK*DOS and your BIOS.

Although SK*DOS may change with revisions, the first 512 bytes have been set aside for linkage to and from the boot program, BIOS, and optional drivers for other routines. The following paragraphs describe this area, with the assumption that we are discussing an SK*DOS which is ORG'ed at $1000.

Startup Area

The so-called 'startup area' contains those locations which are referenced when the system is first booted or restarted. (As mentioned earlier, SK*DOS usually starts at location $1000, and the following discussion assumes that. If your version has been specially ordered for a different address range - because you have no RAM at $1000, for example - then you will have to modify the addresses listed below to suit your configuration.)

$1000 Cold-start entry

$1006 Warm-start entry

These two entry points are JMP instructions which transfer control to the cold- start and warm-start locations of SK*DOS, respectively. These are generally not used, as applications programs use traps to re-enter SK*DOS, but the boot program as well as a monitor ROM (such as HUMBUG) may use these to enter SK*DOS from higher level programs.

$100C Get startup date

This location contains a 6-byte JMP which points to the routine within SK*DOS which asks for the date upon bootup. This JMP is in turn called upon booting. If your system has a calendar IC which can provide the date automatically, you may provide a routine (perhaps as a part of the BIOS) which sets the date automatically rather than asking for it. In that case, the BIOS would substitute a JMP to its own date routine at $100C.

$1012 Time entry point

This location contains a set of three RTS instructions (a total of six bytes) which are called each time SK*DOS opens a file for writing or updating. If your system contains a clock IC, then you may substitute a JMP to a subroutine which gets the current time from the clock and returns it in D5 (this routine must preserve all registers!) In order to squeeze the time into one byte, it is encoded with a resolution of six minutes, where

$00 = no time (invalid time)
$01 = 12:06 - 12:11 (a.m.)
$02 = 12:12 - 12:17 (a.m.)
$03 = 12:18 - 12:23 (a.m.)
. .
$EF = 23:54 - 23:59 (p.m.)
$F0 = 12:00 - 12:05 (a.m.)

Note the distinction between $00 (invalid or no time) and $F0, which is the 6 minutes beginning at midnight.

$1018 OFFINI (four bytes)

This location contains the initial value of OFFINI in SK*DOS.COR, and the modified value after you append your BIOS.

$101C MEMINI (four bytes)

When SK*DOS is booted, it does a memory test to determine how much memory the target system contains, and then stores the highest valid memory address in MEMEND. MEMINI is used to define the top limit of that memory test. It is especially important in those systems which do not properly implement DTACK for nonexistent memory, since in such cases the system may hang up if SK*DOS tests a location above the actual end of memory.

Secondary Driver Area

This section comprises an area for linking to secondary disk drivers, RAM disks, or disk caches. These locations are for secondary disk drivers, and are explained in the User Manual; they are normally filled with RTS instructions, and need not be changed if you do not use such drivers.

Trap Flag Area

This area contains information involving the 68000's exception vector (trap) processing.

$10C0 TRPFLG

This byte, if non-zero, causes SK*DOS to place all of its own exception vectors into lower RAM; if zero, then SK*DOS places only the 'line 1010' vector, which is used to implement $Axxx instructions which call SK*DOS. This byte is also explained in the User Manual.

$10C4-10C7 LATRAP

This location normally contains the address $00000028, which is the address of the 680x0 processor's "line 1010 emulator" trap vector (which is used to steer all $Axxx traps to SK*DOS). Most 680x0 systems have a ROM at address $0000 when they first are started, but remove this ROM and substitute a RAM a specified number of clock cycles later. In this case, SK*DOS will place the appropriate vector into location $0028 during its initialization process, and so the typical user will not have to concern himself with LATRAP (or BETRAP, following).

In some cases, however, the ROM stays permanently mapped, in which case SK*DOS cannot place this (and the BETRAP) vector into the appropriate exception vector location. In this case, the user will have to do three things: (1) place a JMP.L $0 instruction somewhere in free RAM, (2) burn this instruction's address into location $0028 of the ROM so a 'line 1010' trap will steer the processor to the JMP, and (3) put a vector pointing to the *address field* of the JMP instruction into the LATRAP bytes in SK*DOS, thereby telling SK*DOS where to put its own vector; the latter would be done in BIOS in the same way as setting MEMEND or OFFINI.

$10C8-10CB BETRAP

BETRAP is very similar to LATRAP, discussed above, except that it normally contains $00000008, the address of the bus error exception vector. It is used in the same way, and need only be changed by the user if the system's exception vectors are permanently stored in ROM.

Primary Disk Driver Area

Most of this disk driver area will always be used by your BIOS. In each case, the drivers should assume that A4 points to a valid file control block containing the required logical drive, track, or sector number. The addresses listed below are normally filled with RTS instructions. Any bytes not used can be left as is.

$1100 DICOLD Disk driver cold start

Your BIOS will place a JMP instruction at DICOLD which will send SK*DOS to do a cold-start initialization of your disk drivers.

$1106 DIWARM Disk driver warm start

The six bytes beginning at DIWARM will be replaced by a JMP to your BIOS warm start initialization routine.

$110C DIREAD Disk read vector

As before, your BIOS will place a JMP here to go the primary disk read routine.

$1112 DIWRIT Disk Write vector

As before, your BIOS will place a JMP here to go to the primary disk write routine.

$1118 DICHEK Disk check vector

Your BIOS will place a JMP here to go to the disk check routine.

$111E DIMOFF Disk motor off vector

In those systems which require a programmed shutoff of the disk drive motor, this will be replaced by a JMP to the disk motor off routine.

$1124 DIREST Disk restore vector

Your BIOS may place here a JMP to the disk restore routine. This vector would normally be used only by the FORMAT program.

$112A DISEEK Disk seek vector

DISEEK is similar to DIREST but points to the disk seek routine.

$1130 STPRAT Drive step rate

Ten bytes which define step rates for up to ten floppy drives. (Actually, our suggested disk drivers only use the first byte to assign the same step rate to all floppy drives.)

$113A VERFLG disk verify flag

A non-zero number indicates that disk sectors should be verified after being written. This is a global flag for all drives, although it may well apply only to floppy drives in some systems.

$113C DRUSED Drive used table

Ten bytes which describe what disk drives are in use, what type they are, and which also relates the logical and physical drive numbers (see the DRIVE command in the User's manual). Each of the ten bytes corresponds to one of the ten logical drives (0 through 9). The format of each table entry is as follows:

$00 = drive does not exist

Left nybble:

1 = floppy drive controller
2 = hard drive controller
4 = other controller
8 = RAM disk

Right nybble:

The physical drive number on the corresponding controller.

For example, an entry of $12 means drive 2 (the third drive) on the floppy controller.

$1200 WINTAB Winchester/hard disk data table

WINTAB is a 128-byte long table containing data regarding hard disks, their partitions, and other physical data. The first 64 bytes are for hard drive A, the second 64 are for hard drive B. Each group is then subdivided into four 16-byte segments, one for each possible partition on the disk. Each of these 16-byte segments is laid out as follows:

Byte 0: $FF if empty, else $20 through $27. $20 is partition 0 on drive A, $27 is partition 3 on drive B.

Byte 1: The WD steprate code, probably 0 for buffered seek

Byte 2: Sectors per track on hard disk, probably $20

Byte 3: Number of heads, probably 2 or 4

Bytes 4-5: Number of sectors per cylinder = byte 2 times byte 3

Bytes 6-7: Number of hard disk cylinders in this partition

Bytes 8-9: First cylinder number in this partition; must be 0 for partition 0 on either drive. For example, if partition 0 has 100 cylinders, then partition 1 starts on cylinder 100.

Byte 10: last logical track number of partition (1 - $FF)

Byte 11: last logical sector number of partition (1 - $FF)

Bytes 12-13: Cylinder number to park on

Byte 14: Cylinder to start precompensation on, (divided by 4 for the WD-1002-HDO hard disk controller).

Byte 15: empty, for future use.

The 64 bytes for each drive are also written into track 0 sector 0 of the corresponding drive, except that for the first byte only the last 2 bits of the partition code are written. For example, codes 21 or 25 are written as just 01. This is so the hard drives can be physically switched if need be.

The data is put into WINTAB in one of two ways - HDFORMAT does it after formatting, and cold-start does it upon booting. Hence WINTAB should generally be an accurate picture of hard disk data. If there is data in WINTAB, then DRIVE will print out a summary. WINTAB is used primarily by the disk drivers to identify partitions, and to do the conversion between logical track/sector and physical track/sector. It is also used by PARK to decide where to park the heads when asked.

Unlike floppies which have no sector 0, the hard disk drivers allow sector 0 on each track so as to provide the maximum disk capacity. Track 0 sector 0 is special in that it contains WINTAB, and is protected in that the disk write routine in SK*DOS cannot write this sector.

Console Driver Vector Area

The following vectors are used to interface the console drivers of BIOS to SK*DOS. In each case, SK*DOS.COR contains three RTS instructions for each vector; you may leave these if a particular vector is not used, or replace them with a JMP to a routine in the BIOS if used.

$1180 SINITV Serial port initialization

This points to a routine which initializes the serial port. In most cases, the port will actually already be initialized by the boot ROM, so this may not be used.

$1186 STATVE Keyboard status vector (via typeahead, if any)

$11D4 STATV1 Keyboard status vector (1 char only)

These routines should determine whether a character is waiting in the keyboard, and return a zero condition if not. Typeahead will be discussed later.

$118C OUTCHV Output a character to terminal

This routine should output a character from D5 to the terminal.

$1192 INCHV Keyboard input with echo

This routine should input a character from the keyboard into D5, clearing the parity bit to 0 and echoing it to the terminal.

$1198 KINPUV Keyboard input without echo (with typeahead, if any) $11DA KINPV1 Keyboard input without echo (1 char only)

These routines should input an 8-bit character from the keyboard into D5 without echoing to the screen. Typeahead operation is discussed later.

$119E OCNTRL Output channel control

This routine should output a control character from D5 to the BIOS for controlling the output channel.

$11A4 ICNTRL Input channel control

This routine should output a control character from D5 to the BIOS for controlling the input channel.

$11AA MONITR Re-enter monitor ROM

This routine should re-enter the system (monitor) ROM without doing a major reset. In particular, this entry point should not reestablish the monitor exception vectors.

$11B0 MRESET Monitor reset

This JMP should return to the monitor ROM and do a complete reset, including resetting all of the monitor's own exception vectors.

$11E0 KILLV1 Flush typeahead buffer

This routine should empty the keyboard typeahead buffer of all characters and clear all keyboard input flags.

Getting to the above variables

For the sake of simplicity, all of the above variables are listed with their actual addresses in those systems where SK*DOS begins at $1000. When you write your BIOS, you will know where SK*DOS is ORG'ed, and so you can refer to them directly. They should not, however, be referred to by their absolute addresses in normal application programs, since SK*DOS may be ORG'ed elsewhere on some systems. The starting address of SK*DOS is, however, at location DOSORG, which is located at 838(A6). Thus, for example, MRESET can be obtained with

```
DOSORG   EQU      838
         DC       VPOINT     Get A6 pointer
         MOVE.L   DOSORG(A6),A5
```

A5 now points to the beginning of SK*DOS, and the address of MRESET is now $1B0(A5).

SK*DOS I/O Philosophy

The precise details of how the BIOS interfaces with this area of SK*DOS will become apparent later in the manual, and may be observed in the BIOS source code on the accompanying disk. This section will simply describe the philosophy behind the I/O organization of SK*DOS.

As indicated above, there are two sets of disk I/O vectors - the primary and the secondary set. Although the primary set is the main one, most often used for two to four floppy drives, SK*DOS accesses the secondary vectors first. The intent is that the secondary vectors be used for a RAM disk, a disk cache, or for secondary drives. The corresponding routines should simply look at the drive number (which is in the drive number byte of the FCB pointed to by A4 on entry), and perform an immediate RTS if some other drive is being referenced. If this occurs, or if the secondary vectors are still RTS instructions, which is what normally fills these vectors, then SK*DOS goes to the primary vectors which send it to the BIOS to go to the primary drives. Any errors, including invalid drive numbers, are then detected by the primary drivers.

The secondary driver vectors are initially ten RTS instructions. If you install secondary drivers, you should replace the first three available RTS instructions of each set with a corresponding JSR to the driver; the driver should then do one of three things:

a. RTS immediately if the wrong drive number is specified, or

b. If the correct drive is specified, and no error occurs, pull two return addresses off the stack and then RTS back to the calling program with a zero condition, or

c. If the correct drive is specified for the secondary driver but an error occurs, then pull two return addresses off the stack and RTS back to the calling program, but with a non-zero condition, and with the error number both in the error byte of the FCB and also in the ERRTYP byte.

Since the secondary vector area starts out as ten RTS instructions, for a total of 20 bytes, there is room for up to three sets of JSR instructions in each, with the last JSR still followed by one RTS. Hence there can be up to three sets of secondary drivers; if all three do an immediate RTS because their drive number has not been specified, then the last RTS will send control back to SK*DOS, and then to the primary drivers.

In all above cases, all registers must be preserved. On entry, A4 always points to the FCB which requires the disk operation.

The DIREAD and DIWRIT calls to the primary disk drivers are substantially different from the corresponding calls in 6809 SK*DOS. In the 6809 version, the disk read and write routines in the BIOS were very elementary, as SK*DOS itself took care of retries in case of error, counted the number of retries, reported errors, and handled verification on writing. In SK*DOS/68K, it is assumed that the drivers will handle all this. This is a more valid approach, since it is obvious that retries, for example, should be handled totally differently in floppy and hard disk controllers.

CHAPTER 3 - HUMBUG AND BOOT ROMS

Any 68K system needs a system ROM which will properly initialize the system when it is first turned on. In addition, when first installing SK*DOS on such a system, it helps if this system ROM also contains some additional debugging functions which can be used to install SK*DOS.

If your system already has such a ROM, then skip this chapter and go on to Chapter 4. If not, then this chapter describes two such system ROMs which you can use - a fairly complete ROM monitor called HUMBUG (R) which can be used when debugging SK*DOS, and a very simple 'boot ROM' which can be used once SK*DOS is properly installed and working.

NOTE: Although the code for HUMBUG is provided as part of this Configuration Manual, it is not part of SK*DOS. It is provided for *your* use only, strictly as an aid in properly bringing up SK*DOS on your system. If you are licensing SK*DOS for installation on a number of systems, please be aware that **your license does not include HUMBUG.** You may include the 'boot ROM' as part of each SK*DOS you produce under your licensing terms, but that license does not include HUMBUG, which is a separate product and must be licensed separately, if desired.

These two system ROMs are described in the rest of this chapter.

HUMBUG

The HUMBUG.TXT file on your disk contains the code for HUMBUG; you must customize it for your system, and then assemble it into an object code module.

The version of HUMBUG included was designed for a 27128 EPROM which is ORG'd at $FFFFC000. It is not position independent, and hence must be reassembled for different memory locations or ROM sizes.

The code for HUMBUG consists of five parts:

1. A data area containing several variables needed by HUMBUG. Chief among these are

PORADM - the address of the UART or other device used for I/O with the main control terminal

PORADP - the address of the UART or other device used for I/O with the printer

DVECTR - a JMP instruction which points to an auxiliary, user-provided character output routine.

2. 192 bytes of data (48 long words) which contain the 48 exception vectors for the 68K processor. These vectors will all be copied into the lower part of RAM once HUMBUG is run.

3. A jump table of entry points into some of the more useful routines of SK*DOS, useful for any programs you might write. This table, beginning at $FFFFC0C0, is a series JMP instructions (to be accessed by JSR instructions) for the following:

$FFFFC0C0	Cold-start; total restart of HUMBUG.
$FFFFC0C6	Warm-start
$FFFFC0CC	INEEE input keyboard character into D5.B, obey control-S options
$FFFFC0D2	INCH7 input 7-bit keyboard character, with bit 7 = 0
$FFFFC0D8	INCH8 input 8-bit keyboard character
$FFFFC0DE	OUTEEE output screen/terminal/printer character from D4.B, obey control-S options
$FFFFC0E4	OUTCHM output screen/terminal character
$FFFFC0EA	OUTCHP output printer character
$FFFFC0F0	PSTRNG display the string pointed to by A4, up to an $04 delimiter
$FFFFC0C6	OUT4HS display a 4-digit hex number contained in D4, followed by a space
$FFFFC0FC	OUT8HS display an 8-digit hex number contained in D4, followed by a space

4. The main code for HUMBUG.

5. The actual I/O routines which you will have to write to adapt HUMBUG to run on your system. These are described in the following paragraphs.

Your I/O routines will consist of the following parts; see the source code in the HUMBUG.TXT file for examples.

a. MEMRD and PARON. These routines are only needed for those systems which provide parity checking for memory; otherwise simply keep the two RTS instructions as shown.

In a properly designed parity checking memory, the parity circuitry will generally be disabled whenever a reset is performed. To properly initialize parity circuits the first time the computer is turned on, you must first read each location of memory and then rewrite the original number back in. This makes sure to write back the correct parity bit. Then you must actually turn on the memory parity circuit. This requires two separate routines - MEMRD to read all of memory and write each location back, and PARON to enable the parity circuits. HUMBUG calls MEMRD and PARON, in that order, when first started, and PARON only after each succeeding cold start (reset). If a memory error is detected during MEMRD, you may use the PSTRNG and OUT8HS routines to print an error message.

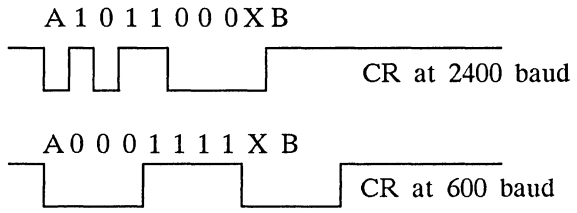
These two routines may have to change some of the exception vectors. For example, MEMRD may have to modify the bus error exception vector if it might test nonexistent memory. Depending on what kind of an interrupt your memory parity circuits generate, you will also wish to change that vector as well. Finally, SK*DOS itself should print out an appropriate error message in case of a parity error; the easiest way to achieve this is to change the appropriate message in the ERRCODES.SYS file after SK*DOS is up and running.

b. INIPOR is a routine which initializes the console and printer terminal ports. The routine shown is for a DUART located at address $E0010, but obviously you may have to make major changes in this routine.

The first part of INIPOR initializes the printer port, which in our case is simply the second half of the DUART. In this case, it is initialized at 9600 baud, 8 bits, 1 stop bit and no parity. The second part of INIPOR initializes the main terminal port, which is the first half of the same DUART. In this case, however, the program determines the baud rate of the terminal and sets the

DUART baud rate accordingly. This requires that the user type in a CR (carriage return or $0D character) immediately after turning on the computer. This is done as follows:

The DUART is initially configured for 2400 baud, but suppose that the user's terminal actually works at 600 baud. Then the following two waveforms show the $0D that the UART expects to get, and the 600-baud $0D that it actually gets (A is the start bit, B is the stop bit, and X is the parity bit which is removed by the routine and replaced by 0):



Keeping in mind that the X parity bit is replaced by a 0 in the input routine, the DUART interprets a 600-baud CR as a binary pattern of 01111000 or $78, instead of its correct $0D. The lookup table at BAUTAB is then used to convert the $78 actually received into the appropriate code ($55 in the case of the DUART) to set the port to 600 baud.

c. INCHEK is the routine which checks for a character coming from the keyboard, without actually getting that character. It returns a 'zero' condition if no character is present.

d. INCH8 waits for a character, and then returns it in D5.

e. OUTCHP outputs a character from D4 to the printer.

f. OUTCHM outputs a character from D4 to the main terminal.

g. FLBOOT is the boot routine which reads sector 1 of track 0 of the disk into memory at location $0800, and then jumps to it. In case of an error, it prints an error message and returns to the warm-start reentry of HUMBUG.

h. RETDOS is a routine which returns to SK*DOS when the RD command is typed in HUMBUG. It simply checks that there is a JMP instruction at cold-start, and then goes to the warm-start entry point. This routine is provided here just in case it is necessary to ORG SK*DOS at some unusual address in your target system.

Note that neither the console routines nor the disk routines use interrupts in any way. Although ultimately it may be desired to implement a type-ahead buffer or other interrupt-driven I/O, the ROM monitor is not the place to do it since (a) speed is not needed here, and (b) debugging a ROM is difficult enough without introducing added complexities.

HUMBUG Commands

Once HUMBUG is operating, it responds to two-letter commands from the keyboard. For example, the HE command prints a 'help' message which gives a short listing of available commands.

Many of the HUMBUG commands, such as memory dump commands, require a starting and ending address for proper operation. These commands prompt for this pair of addresses with a FROM ... TO ... You may enter addresses in free form, and need not enter initial zeroes. For example, address $00000123 could be entered as 00000123, or 0123, or just 123. (Each number entered into HUMBUG should be followed by a space to indicate that it is finished.) The pair of addresses entered by the user is called the "FT" pair, and is stored for possible reuse later.

Any valid hex address is acceptable as a response to the FROM ... TO ... prompt. If a CR is entered instead of the FROM address, then the current command will use the last FT pair previously entered. Any other character (except a space) will cancel the execution of the command and return to HUMBUG command entry mode.

To use HUMBUG, simply turn on the computer and press CR once or twice until the initial signon appears. Then use one of the two-character commands below:

AD - Formatted ASCII dump. The specified area of memory is dumped to the output device, sixteen bytes to a line. Each line is identified with its starting address. ASCII codes of 7E, 7F, and 00-1F are printed as a period, and the most significant bit (parity bit) is ignored.

AI - ASCII Input. The AI command allows the direct input of ASCII data from the keyboard into any area of memory. All text following the AI is inserted into the memory area defined by the FT pair. If the memory area set aside is too small to hold all the text entered, or if the text is not properly stored (due to nonexistent or defective memory), either you will get a BUS ERROR error message, or your screen will start outputting the word ERROR immediately after the last possible character has been stored. The only way to get out of the AI mode is by the control-S/CR combination, or by pushing RESET. When this is done, the FT pair will be changed to reflect the amount of memory actually filled by the AI, so that a following AO or HD command would output exactly the same data as entered by AI.

AO - ASCII Output. Following this command, the contents of the memory area defined by the FT pair is output to the screen. This is normally used to output ASCII text. The AI-AO combination is primarily intended for testing.

BP - Print Breakpoints. HUMBUG allows up to four breakpoints to be set at the same time. The BP command prints out the addresses of the current breakpoints, and the operation codes of the instructions at those breakpoints, so that the user does not forget their locations.

BR - Breakpoint set/reset. The four possible breakpoints are numbered 1 through 4, and can be individually set or reset. When the system is first turned on, all breakpoints are erased; subsequent RESETs do not erase the current breakpoints. The typical BR command has the following form:

```
BR NUMBER: n ADDRESS: addr
```

where the computer's responses are underlined. n is the number of the breakpoint you wish to set or reset; addr is the new address of that breakpoint. Entering a new address, or hitting CR or any invalid entry for addr, will cancel the old breakpoint number n.

CO - Continue. After a breakpoint is encountered in a program, or after a single-step execution, the program being tested may be continued with the CO command. After a breakpoint, the breakpoint should be removed with the BR command before hitting CO; otherwise the break will be executed again and the program will not go on.

CS - Checksum. This command prints a 16-bit checksum of the memory area defined by the FT pair. This is primarily intended to check whether a program or data has been properly loaded, or whether it has been changed.

FD - Floppy disk boot for 5" disk systems.

FI - Find. FI will print out all addresses in the area of memory defined by the FT pair which contain a specified one-through five-byte constant. The typical command sequence is

```
FI HOW MANY BYTES? n FIND WHAT? dd
FROM addr TO addr
```

where computer responses are underlined. n is the number of bytes to be found, dd are 2 through 10 hex digits representing the 1 through 5 bytes to be found, and addr are the two FT addresses specifying the address range to be searched.

FM - Fill memory. This command allows a specified area of memory, defined by the FT pair, to be filled with a specified byte.

HA - Hex and ASCII dump. Combination of the HD and AD commands, which prints the contents of memory in both hex and ASCII. This command requires a terminal width of more than 64 characters; in narrower displays, the HD and AD commands must be used instead.

HD - Hex Dump. Prints a hexadecimal dump of the area of memory defined by the FT pair. Sixteen bytes are printed per line, with each line preceded by the address.

HE - Help. Prints a listing of all HUMBUG commands.

JS - Jump to System program. This command jumps to the address specified after the JS, with the CPU in system (supervisor) state.

JU - Jump to User program. This command is similar to JS, but enters the program with the CPU in user state.

LO - Load S1 - S9 Motorola binary format from main keyboard.

MC - Memory compare. This command compares two specified memory areas byte-by-byte, and prints out memory contents for each byte which is different in the two areas. Prompts ask for the FT pair for the first area, and for the starting area of the second.

ME - Memory examine and change. This command allows you to examine the contents of memory on a byte-by-byte basis, and enter new data if desired. When you type in ME, followed by the address to be examined, HUMBUG will display the current contents of that address and wait. You may now type in one of the following:

a new byte of data a space to go to the next byte
an up arrow to go back to the previous byte
anything else to quit

MO - Move memory. This command allows the contents of the memory area specified by the FT pair to be moved to another memory area. Memory data can be moved to higher or lower addresses, and the new area can overlap the original area. Moving is done in the correct way so that no data is lost even on overlaps.

MS - Memory Store. This command is similar to ME, but stores data without first reading it out, and without verifying that it was properly stored.

MT - Memory Test. Does a simple memory test on the memory area defined by the FT pair. If memory is OK, it prints a plus sign and returns to HUMBUG. If memory is bad, it prints the address of the bad location, a hex number representing the bad bit, and the actual contents of that location at the time it failed the test. (This is a non-destructive test of memory since the previous contents of each location are restored. If, however, a memory test is done of I/O locations, it is possible that false I/O operation may occur, or that I/O devices may not be properly initialized.)

RC - Register Change. Allows you to modify the contents of the CPU registers displayed by the RE (Register Examine) command. When you type RC, HUMBUG will respond with REG:, which you should answer with the code for the register to be changed. HUMBUG then enters the ME mode at the location where that register is stored. You may then examine or change the register contents, as desired. Note, however, that register A7 cannot be directly changed. Instead, you must change either US (user stack) or SS (system stack).

RD - Return to SK*DOS. This command returns to your DOS. Caution - do not use the RD command unless SK*DOS has already been booted and run.

RE - Register Examine displays the current CPU registers. The Data and Address registers are displayed first, with the remaining registers below. An RE printout is automatically performed following a single-step or upon encountering any breakpoint. Note that the display for A7 depends on which state is currently reflected in the status register; changing the current state will also generally change the A7 display.

SS - single-step. Perform the next instruction of the program being tested. SS uses the register contents printed by the RE command; hence the SS command cannot be used to start single-stepping until after a prior breakpoint or single-step has been performed. When an SS is performed, HUMBUG prints out several lines: the first line prints out the address of the instruction to be performed, while the other lines print out the RE dump after the instruction has been performed.

ST - Start single-stepping. Since SS cannot be performed until after a breakpoint or previous single step, the special ST command is included to perform an initial single-step if the breakpoint is not used. ST prompts for the address of the first instruction to be single-stepped, and then executes it in exactly the same way as the SS instruction.

WD - Winchester Disk boot. Obviously only useful for systems which contain a hard disk, this command is used for booting directly from it. Note that you may have to customize this routine to fit the particular hardware installed in your system. The WD command also initializes the DRUSED table so that the hard disk becomes drive 0 and a floppy drive becomes drive 1; you may wish to change this convention.

!! - Monitor reset command. HUMBUG does not normally erase breakpoints except at the first power up; other resets omit this step. The !! command does a complete reset, exactly the same as at power up. In general, this is a command which will not be commonly used, and hence has been assigned a non-standard command code.

HUMBUG I/O Control

A serial port is normally used for all monitor input and output.

In addition, HUMBUG can provide an output to a second serial (printer) port or to a user-written output routine in RAM or EPROM.

Any time that the monitor is looking for commands, or any time that INEEE or OUTEEE are called, HUMBUG checks the control port for a control-S break character arriving from the keyboard. When a control-S is detected, HUMBUG rings the bell (control-G) and halts all current I/O.

When I/O is halted, HUMBUG waits for one more character which is used for controlling monitor ports. This control character can be one of the following:

Carriage Return (CR) - this cancels the current program and forces a return to the monitor.

C - turns the main (control) port on or off.

P - turns the printer port on or off.

X - does the same for a user-written port routine.

W - turns the wait (pause) mode on and off. When the wait mode is on, output will stop every 15 lines to allow it to be read on a CRT terminal.

Any other character is ignored.

The C, P, and X characters toggle their corresponding ports; if a port is on then it goes off, if it is off then it goes on. Since these characters are not echoed or returned to calling programs, ports can be turned on and off in the middle of input or output.

STATUS is a byte in the monitor scratchpad which indicates which output devices are currently active. The bit assignment in STATUS is as follows:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Port P	Port C		Port X	Wait			

The control-S / CR combination allows many stuck programs to be aborted without reaching for the RESET button.

HUMBUG provides facilities for a user port called Port X, which is toggled on and off using the combination of control-S followed by X.

When port X is selected, HUMBUG does an indirect JSR to a user written routine via location DVECTR in the scratchpad RAM. This location is initialized to point to an RTS during restart of the monitor.

To access a user-written routine via port X, a vector must be placed into DVECTR to point to this routine. Output may then be started and stopped with the control-S / X combination. (Port X will also be disabled automatically when a user program returns to HUMBUG via either the cold-start or warm-start re-entry points.)

Although only one vector can be placed in DVECTR at a time, if two routines need be driven at the same time, this vector can point to another program which in turn steers output to the two routines.

BOOT ROM

The BOOTROM.TXT file on your disk contains the code for a boot ROM which may be used to boot SK*DOS in a system which does not have another ROM. This code is essentially the boot portion of HUMBUG, without all of the extra debugging tools. You must customize it for your system, and then assemble it into an object code module; customization is similar to that for HUMBUG.

As written, the boot ROM code provides for a boot-up menu which gives you the choice of booting from either a floppy disk or a hard disk. It also provides facilities for a disk boot for an alternate operating system, and for parking the head on a hard disk. You may wish to delete portions of this code.

If you are licensing SK*DOS for resale, this boot ROM is part of your SK*DOS package, and you may distribute it as a part of each licensed SK*DOS package you sell. Please **do not** accidentally release the HUMBUG program.

CHAPTER 4 - BIOS

The BIOS, or Basic Input/Output System, is the main interface between SK*DOS and the hardware of the target computer. It contains the programs which directly interface to the terminal (keyboard and screen), and to the disk drive, as well as a few other, smaller routines.

Chapter 2 has already explained the entry points into the BIOS, and how the BIOS is tied into the main body of SK*DOS. This chapter will therefore cover only implementation details of the BIOS itself.

The BIOS will normally be ORG'ed just above the top of the core portion of SK*DOS. If you are unsure where SK*DOS ends, set the ORG at a high address, such as $C000. Once the system is running, you may then use LOCATE to determine the top address of SK*DOS.COR, and reassemble with a new ORG.

The BIOS code on your configuration disk is divided into the following major portions:

PART 1-A. Driver Select

The sample BIOS contains both floppy and hard disk drivers, and this part selects the appropriate drivers. SK*DOS enters this code with A4 pointing to a file control block. Location FCBDRV of that FCB contains the logical drive number, while FCBPHY contains the physical drive number (which SK*DOS gets from the DRUSED table). This part of the code simply looks at FCBPHY to determine whether the left nybble is a 1 (floppy) or 2 (hard) disk code, and steers the machine to the appropriate drivers.

PART 1-B. Floppy Disk Drivers

This section consists of the following:

a. The Read entry point. This is essentially equivalent to the SREAD function of SK*DOS. It receives as argument the address, in A4, of a file control block which contains the drive, track, and sector numbers of a sector to be read into the FCB. It selects the drive, seeks to the track, and reads. If an error occurs, then it does a restore to track 0 and tries again, up to a total of four times.

b. Write entry point, quite similar to the read routine except for the addition of a disk verify if the VERFLG is nonzero.

c. Error processing routine which converts FDC error numbers into SK*DOS error numbers.

d. The DDRIVE routine which selects the specified drive, sends the drive number, side, and density bits to DLATCH, and turns on the motor.

e. One-second time delay to give the motor time to come up to speed.

f. A disk restore routine which restores the selected drive's head to track 0.

g. A WAIT routine which waits 30 microseconds or more. This is needed since the FDC requires a delay after any command is sent to its command register.

h. WNBUSY waits for the BUSY bit in the FDC's status register to go off after a command is finished executing.

i. SIDENS is a routine which analyzes the current drive, track, and sector to determine which side or density to use, and then places the appropriate bits into D6 in preparation for their being placed into DLATCH.

j. DSEEK which sends the commands to the FDC to seek to the specified track.

k. DREAD which is the actual disk read routine. In order to save time, and allow both single- and double-density operation under programmed I/O control, A4 and A6 point to the data and status registers of the FDC and address-register indirect addressing is used. As written, this routine does not count bytes, nor does it do a timeout. If a disk is not inserted into the specified drive, this routine just waits. We decided to use this approach for two reasons - to save time in the loop and guarantee that programmed I/O could handle double-density operation, and also to avoid problems when some programs abort if a disk is not present. In this way, the system simply waits if a disk is not present, which gives the user the time to insert a disk and continue normally.

l. DWRITE, similar to DREAD, which does an actual write.

m. DVERIF, which is used to verify that a sector just written is readable. Note that DVERIF does not actually check that the data on the disk matches the data written, only that it can be read. Although you can obviously change this, we decided to save the time that would otherwise be necessary to individually check every byte of data.

n. CHKRDY checks whether the specified drive is ready or not. Actually, since most 5-1/4" drives do not have a ready line, this routine simply checks for a valid drive number. CHKRDY tests the drive entry in the DRUSED table to determine whether that drive exists. If this entry is not valid, then it is assumed that this drive does not exist and the routine generates a not ready error.

PART 1-C. Hard Disk Drivers.

The hard disk drivers are totally separate from the floppy drivers, and consist of the following:

a. HARDRD actually reads a disk sector from the disk.

b. HARDWR writes a sector to the hard disk.

c. CHBUSY waits if the disk drive is busy, but aborts with an error message if it takes more than about 5 seconds.

d. An error processing routine to handle hard disk errors and convert them into SK*DOS error numbers.

e. LPCONV is a routine which converts a logical track and sector number into a physical track, head, and sector number, and stores them in the disk controller registers. This routine uses data from the WINTAB table to determine the actual organization of the disk.

PART 1-D. Hard Disk Initialization

This part is called upon cold-start of SK*DOS. Aside from initializing some of the hardware, its main purpose is to read the WINTAB data from track 0 sector 0 of either or both hard disks, and store in WINTAB. It is essential to make sure to test for the

existence of a hard disk, and not get hung up if no hard disk exists.

PART 2. Console Drivers

This section contains the code for initializing and using the main serial port through which SK*DOS is controlled. It also happens to contain the code to return to the monitor in response to the MON and RESET commands. It consists of the following:

a. SERINI is a routine called at cold and warm start whose purpose is to initialize the console port. In HUMBUG or our boot ROM, as in most instances, initialization is handled by the startup ROM, and so the SERINI shown only contains an RTS instruction.

b. STAT and STATM check whether a character is waiting in the keyboard input. The sample code shown samples the keyboard ready bit of the DUART and returns a zero condition if no character is waiting in the keyboard buffer. It also defaults to the same routine for both STAT and STATM; a fuller description of typeahead follows later in this chapter.

c. OUTEEE outputs the character in D5 to the screen. Note that this routine also calls MOFAST to slowly turn off the disk motor.

d. INEEE uses KINPUM to get a character and then OUTEEE to echo it if the INECHO flag is on.

e. KINPUT and KINPUM actually load a character from the keyboard, and strip the parity bit. Note that the routine shown also calls MOSLOW to slowly turn off the motor. Note that both entries use the same routine in the sample code; a fuller description of typeahead follows later in this chapter.

f. MONIEX and RESETX return to the boot ROM in response to the MON and RESET commands.

g. KILLTA flushes the typeahead buffer, if any. A fuller description of typeahead follows later in this chapter.

PART 3. Get Date Routine

GETDAT is used only if the system has a clock-calendar IC. It is called when SK*DOS is being booted, and reads the date from the clock rather than asking for it from the keyboard. Do nothing if no clock IC is present.

PART 4. Get Time Routine

This part is used only if the system has a clock-calendar IC, and consists of two subparts. INTIME is called each time SK*DOS opens a file for writing or updating, reads the time from the clock IC, packs it into D5 as a single byte, and then returns. GETDT is called by the GETDNT function, and returns the date in D5 and time in D6. Incidentally, note that the clock itself is set with the TIME command.

PART 5. OFFSET Calculation.

DRVEND is an EQUate which is set to the end of the drivers, and which is then placed into OFFINI to signal the top of BIOS to SK*DOS.

PART 6. Vectors which connect SK*DOS with the BIOS

ORG statements and JMP instructions are overlaid over SK*DOS vectors to steer SK*DOS to the appropriate routines in the BIOS. An END $1000 statement then appends a transfer address to the BIOS, so that when it is finally APPENDed to SK*DOS the boot routine will know where SK*DOS begins.

Note that three of the vectors - those for STATV1, KINPV1, and KILLV1 - are commented out in the sample driver. Since the sample driver does not use interrupt-driven keyboard I/O and a typeahead buffer, these three entry points are not needed. In the SK*DOS code itself (not shown), the STAT and KINPUT vectors are actually JMP instructions which steer the program to STATVE and KINPUV instead, while the KILLV1 vector is normally an RTS instruction. Thus commenting-out the last three vectors simply disables the typeahead flushing, and defaults the other two functions to their primary routines.

PART 7. Miscellaneous

Finally, part 7 has miscellaneous functions which might pertain to a particular system. In the sample BIOS, for example, a different set of default floppy drive steprate codes is overlaid over the STPRAT table to compensate for the fact that the WD 1772 disk controller used has different codes from those used for other Western Digital disk controllers.

The sample BIOS has two features which are not absolutely essential, but which are very useful. First, it can read and write both single- and double-density disks. Although single-density is not expected to be common, it is present so that 68K SK*DOS will be able to read 6809 SK*DOS disks. Second, it can also double-step; that is, it can skip alternate tracks so that a 40-track 5-1/4" disk can be read on an 80-track drive. (It can also write such disks, but because of the narrower track width, a 40-track drive will often not be able to read a disk written on an 80 track drive.)

The selection of density and stepping is entirely automatic. The BIOS disk routines maintain two tables, one entitled D0TABL for track 0, and the other entitled DSTABL for the remaining tracks. Each of these tables contains four bytes, one for each of four drives. The rightmost two bits are used to select the density and stepping for that drive. Each time that the read routine gets an RNF (record not found) error from the FDC, it increments the drive entry in one of these tables and tries again. After four tries it will therefore try all four combinations of density and stepping in an effort to read the sector.

Note also that the sample BIOS disk routines do not count the bytes as they read or write a sector. Instead, they use the BUSY bit in the status register to determine when to stop reading or writing. This is an important feature, as it allows the BIOS to read sectors which are longer or shorter than 256 bytes. For example, the TOMSDOS and FROMSDOS utilities require the ability to read and write 512-byte sectors; this would not be possible if the BIOS counted bytes. Likewise, future enhancements to SK*DOS may include the ability to read and write longer sectors.

Interrupt-driven I/O and keyboard typeahead

Although the sample BIOS shown uses strictly programmed I/O, many people find interrupt-driven I/O to be preferable. Although most SK*DOS/68K implementations do not use interrupt-driven I/O, provisions do exist for providing this option. This section describes this in greater detail. Note that this is strictly optional; if you do not wish to provide it, then you need do nothing.

Interrupt operation of an input port makes it possible to maintain a type-ahead buffer which stores keyboard characters that are input until they are needed by SK*DOS. While such operation makes using a DOS somewhat smoother, it does complicate matters.

Quite aside from the difficulties associated with interrupts, a keyboard typeahead buffer requires special handling in three special situations.

First, at the end of every line it outputs, SK*DOS checks to see whether a character is waiting in the keyboard. If so, then SK*DOS inputs it to see whether it is an ESCape, which is commonly used to pause or abort a program. The problem with this is that this empties out the typeahead buffer of characters intentionally entered by the user. Thus there must be some way for SK*DOS (or a user program such as Basic) to check for and accept a keyboard character without actually removing it from the queue.

Second, under some conditions, special characters (such as a break or escape character, or perhaps a control-C, must be recognized by software and acted on before other characters perhaps still waiting in the buffer. Thus there must be some way for SK*DOS (or a user program such as Basic) to look at the very last character entered, omitting anything entered previously.

Third, it is sometimes necessary to flush the typeahead buffer to make sure that it is empty of any strays. This is especially important when a program is asking for a Yes or No answer before doing some potentially destructive action. Any stray Y waiting in the buffer might cause the action to go ahead when you really want to stop it.

We therefore added three extra entry points in the BIOS to handle these special cases. First, there is the special routine KILLTA, which simply erases the typeahead buffer when called.

Then there are two different status routines, and two different keyboard input routines. One set of routines (STAT and KINPUT) work only on the last character entered, while the other set of routines (STATM and KINPUM) work via the typeahead buffer. If characters are being rapidly entered from the keyboard, they are stored in the typeahead buffer and input, in their correct sequence by the latter set of routines. The former set, on the other hand, allow checking for and inputting just the last character. In this way, the last character entered is "brought to the head of the line" so it can be checked for control-C or other break characters.

Since the typeahead code is very hardware-dependent, we will provide a general outline below which will have to be heavily customized for your particular situation.

First, there is the typeahead buffer and its pointers:

```
KBBUFF   DS.B 64     TYPEAHEAD BUFFER
KBIPTR   DS.W 1      PTS TO NEXT CHAR IN
KBOPTR   DS.W 1      PTS TO NEXT CHAR OUT
```

In this case, we used a 64-character buffer which was placed at the end of the BIOS, plus two pointers: KBIPTR points to the location in the buffer where to put the next character, while KBOPTR points to the next character to be taken out. Both of these are word variables, as they are used as offsets modulo 64.

In addition, there is also a 1-character buffer and its flag for just the very last character:

```
INQCHR     EQU $FF0C9F
INQFLG     EQU $FF0C9E
```

In this particular implementation, these also happened to be used by the HUMBUG monitor, and so EQUates were used in the BIOS. INQCHR holds the last character received from the keyboard, while INQFLG is a flag which is set to 1 whenever a character is put in, and to 0 when the character is read out.

The above locations are initialized by the code

```
KILLTA     CLR.W KBIPTR
TYPEAHEAD BUFFER EMPTY
           CLR.W KBOPTR
           CLR.B INQFLG   LAST CHAR EMPTY
           RTS
```

This routine is called by SERINI, but it is also a self-contained routine which is callable from SK*DOS (by the FLUSHT DOS call), and the KILLTA entry point is shown in the BIOS code on your configuration disk.

The BIOS now contains an interrupt service routine which is called each time the hardware detects a pressed key on the keyboard. Assuming that the character is in D5, the following code places it into the buffer:

```
KBSAVE   MOVE.B D5,INQCHR      SAVE THE CHAR
         MOVE.W KBOPTR(PC),D7 OUTPUT PTR
         SUB.W KBIPTR(PC),D7 SUBTRACT
                              INPUT PTR
         AND.W #$3F,D7        MOD 64
         CMP.W #1,D7          FULL?
         BEQ.S KBFULL         YES MEANS
                              BUFFER FULL
         MOVE.W KBIPTR(PC),D7 WHERE TO PUT
         LEA KBBUFF(PC),A5    BUFFER
         MOVE.B D5,0(A5,D7.W) PUT CHAR
                              INTO IT
         ADD.W #1,D7          BUMP POINTER
         AND.W #$3F,D7        MOD 64
         LEA KBIPTR(PC),A5
         MOVE.W D7,(A5)       RESTORE PTR
KBSETF   MOVE.B #1,INQFLG     SET THE FLAG
         RTS

* IF BUFFER IS FULL, CHECK INDIVIDUAL
* CHAR FLAG TOO. IF SET, THEN THERE'S
* CHARACTER OVERFLOW SO BEEP
KBFULL   TST.B INQFLG         CHECK IND FLAG
         BEQ.S KBSETF         PROBABLY IS OK
*    Some code to beep the terminal
         RTS
```

The above procedure places the incoming character into both the 64-character type-ahead buffer and the 1-character "last character" buffer. If both are already full, however, it still places the character into the last-character buffer but also beeps the terminal to signify character overflow.

The BIOS now contains two different keyboard status routines: STATM checks whether the typeahead buffer has a character:

```
STATM    MOVEM.L D7-D7,-(A7) CHECK KBD
MOVE.W KBOPTR,D7
         CMP.W KBIPTR,D7   TYPEAHEAD EMPTY?
         MOVEM.L (A7)+,D7-D7
         RTS                        AND RETURN
```

This routine returns BNE status if there is a character, or BEQ if there is not.

STAT, on the other hand, checks only whether there is a last-character:

```
STAT    TST.B INQFLG        CHECK KEYBOARD
        RTS                 AND RETURN
```

There are also two different character input routines. The main routine KINPUM takes its characters from the 64-character typeahead buffer:

```
KINPUM  MOVEM.L D6/A6,-(A7)  PUSH
KBITYD  MOVE.W KBOPTR(PC),D6 GET OUTPUT
                                    POINTER
        CMP.W KBIPTR(PC),D6  CHECK
                            AGAINST INPUT
        BEQ.S KBITYD        EMPTY IF SAME
        LEA KBBUFF(PC),A6   ELSE POINT TO
                             BUFFER
        MOVE.B 0(A6,D6.W),D5 GET NXT BYTE
        ADD.W #1,D6           BUMP POINTER
        AND.W #$003F,D6       MOD 64
        LEA KBOPTR(PC),A6
        MOVE.W D6,(A6)        RESTORE IT
        MOVEM.L (A7)+,D6/A6
        CLR.B INQFLG          CLEAR FLAG
        RTS                   AND RETURN
```

If there is no character in the buffer (signified by KBIPTR and KBOPTR being the same), then this routine waits for it. It then updates KBOPTR to indicate that the character has been taken, and also clears INQFLG to signify that the last-character buffer is also empty.

The second character input routine KINPUT checks only the last-character buffer:

```
KINPUT  TST.B INQFLG      CHECK KEYBOARD
        BEQ.S KINPUT      WAIT IF NOTHING
        MOVE.B INQCHR,D5 THEN GET CHAR
        CLR.B INQFLG      CLEAR FLAG
        RTS               AND RETURN
```

One other item which must be taken care of is to provide code which will enable and disable the interrupt-driven keyboard routines as the user moves between HUMBUG and SK*DOS. The BIOS has the following storage location which is used to hold the appropriate interrupt vector address; the sample program saves the level 5 interrupt vector from location $0074:

```
L5ADDR  DS.L 1 REMEMBER HUMBUG LVL5 ADDR
```

When the user goes from HUMBUG to SK*DOS, either upon booting or with the RD command, SK*DOS calls SERINI to initialize I/O ports. We then place the following code into SERINI:

```
        MOVE.L $0074,A6   OLD LVL 5 PTR
        MOVE.L A6,L5ADDR  SAVE IT
        LEA KBISS,A6      POINT TO OUR
                         INTERRUPT ROUTINE
        MOVE.L A6,$0074
```

This code saves the existing HUMBUG level 5 interrupt vector into L5ADDR, and replaces it with a pointer to the SK*DOS interrupt routine, called KBISS in our example.

Going the other way, both the MONITX and RESETX routines in the Bios now need the following code to restore the HUMBUG level 5 vector upon exit from SK*DOS:

```
        MOVE.L L5ADDR,A6
        MOVE.L A6,$0074          RESTORE
                                HUMBUG VECTOR
```

It is essential to integrate the SK*DOS interrupt processing with that of your monitor, whether it be HUMBUG or something else. When SK*DOS returns to HUMBUG (with either the MON or RESET command), or returns back to SK*DOS (with RD), the transition has to be smooth so that the user does not lose control. In the sample implementation, we partially solved the problem by using the same locations for the one-character buffer INQCHR and its flag INQFLG. SK*DOS uses its own interrupt service subroutine (ISS), and HUMBUG uses its own as well.

By sharing the same locations, however, each program can use a character obtained by the other's ISS. This is **very important**, because there is another way in which the user could drop back from SK*DOS into HUMBUG: by using the TRACE*** command. While debugging an application program, he might be constantly moving back and forth between the two, giving commands to HUMBUG and suddenly executing a part of an application program which uses an SK*DOS input routine, then immediately returning to HUMBUG. Aside from modifying HUMBUG so it switches back and forth between two sets of interrupt vectors, using the same set of locations for INQCHR and INQFLG is the next easiest solution.

When SK*DOS executes the TRACE*** command, it sets an internal flag which disables KINPUM and STATM, and makes it use KINPUT and STAT instead. In other words, from that time until the system is rebooted, the typeahead buffer is disabled. This prevents characters from accumulating in the typeahead buffer while HUMBUG is being used to debug a program; if this weren't done, then all of these HUMBUG commands would suddenly reappear on the SK*DOS command line the next time HUMBUG returned to SK*DOS.

CHAPTER 5 - THE SUPER BOOT PROGRAM

The suggested process for booting SK*DOS from a floppy disk is as follows:

1. The boot program (either resident in the monitor ROM or typed in from the keyboard) is used to load the super-boot program into memory. This boot program is very simple because the super-boot is always in the same place on the disk - track 0 sector 1.

2. The super-boot program then in turn loads SK*DOS into memory. The super-boot can be quite a bit more complicated because (a) it has to find SK*DOS on the disk, and (b) the rest of the disk might be single- or double-density or single-or double-sided. (For an explanation of the way SK*DOS is stored on the disk, see the description of the binary file format in Chapter 13 of the User's Manual.)

[A short detour: Instead of using a two-step booting process as described here, it is possible to boot in one step, similar to the hard disk procedure, by having the boot ROM search the disk directory for the SK*DOS.SYS file, read it from disk into memory, and directly jump into it. This makes the boot ROM routine more complex, but avoids the problems of the super-boot. The disadvantage is that the one-step procedure can only be used with SK*DOS.SYS, whereas the two-step process is more general; it can be used to boot other operating systems, or perhaps even bypass an operating system and boot directly into an application program. It does not even require the same disk format. End of Detour.]

The super-boot is placed on the disk by the FORMAT program; hence it is really part of FORMAT, and you can examine it at the end of the FORMAT source code on the Configuration Disk. This chapter discusses it separately, however, simply because it really is a different program, and it is easier to describe it apart from FORMAT.

The super-boot should be written in position-independent code, even though at first glance this does not seem necessary. Although you know where the boot will place it (or can force it to load at a fixed location if you write your own boot ROM), nevertheless the super-boot is assembled as part of FORMAT. Hence it is assembled at a different location from that at which it will ultimately load. Position-independent code solves that difficulty, although it sometimes makes the program longer. The super-boot attempts to save space by using two address registers as pointers, one to the beginning of the super-boot, the other to DLATCH.

Disk space is a problem with the super-boot. In order to stay compatible with the format established for 6809 SK*DOS, only sectors 1 and 2 of track 0 are available for the super-boot, and the boot ROM normally only loads sector 1. If the super-boot does not fit entirely in sector 1 (which it usually doesn't), then the part in sector 1 must itself load the second sector into memory. Hence the sector-read routine must lie entirely within the first sector. In writing such a super-boot program, you must make absolutely certain that all the routines needed to load the second sector (including any routines needed to handle errors) are entirely contained within the first sector. The end of these essential routines is indicated with a comment in the listing. If this is not possible for a particular system, then the boot ROM must be extended to load all of SK*DOS directly.

In order to avoid having to search the disk directory to find SK*DOS, the super-boot program usually has the logical track and sector address placed into bytes 5 and 6 by the LINK program. This is done in the variable FIRSTS in the sample program. Note that the program initializes this to 00 00, but goes to an ERROR routine if it is still 00 00 at the time of execution. SK*DOS can never be located at track 00 sector 00, and so a value of 00 00 indicates that the LINK program was never executed.

It is assumed that the super-boot is executed directly after being loaded by the boot; hence, the drive motor is on and the correct drive has been selected. Hence the super-boot can omit these steps.

SK*DOS/68K is somewhat different from 6809 SK*DOS in the matter of disk density. On 6809 systems, track 0 was always single-density, even if the rest of the disk might be double-density. Hence the super-boot usually had to read both single- and double-density sectors. But SK*DOS/68K is now a few years later, and it is safe to assume that designers using an advanced processor like the 680x0 will not limit their disk controllers to single-density. Hence SK*DOS/68K assumes that most disks, and definitely the boot disk, will be double-density throughout. (For compatibility, however, the BIOS disk routines can read both single and double density).

Nevertheless, just in case you need to change that convention, this super-boot program uses two locations which must be preset by the FORMAT program when it places the super-boot on the disk:

DIDENS indicates the disk density, and should be 0 for single density, non- zero for double density.

SECSID indicates the number of sectors per side of each track after track 0. The default for single density is 10 (or $0A), and 18 (or $12) if the disk has been formatted in double density.

Like the BIOS, the super-boot also has provisions for double-stepping. Hence it can boot a 40-track disk on an 80-track floppy drive.

While writing and testing the super-boot program, you may wish to remove the JMP (A5) instruction (three lines after DONE) and substitute a return to your monitor. This is the instruction which would normally jump to the beginning of SK*DOS; during the beginning stages your SK*DOS, though loaded into memory, may still not be ready to be run and so it might be dangerous to jump directly to it.

IMPORTANT NOTE: It is essential that FIRSTS, the two bytes which tell the super-boot where to find SK*DOS, be in location 0005 and nowhere else. The reason is that LINK, the program which is used to put this track - sector information into the super-boot, is written for location 0005. It will put the location of SK*DOS into byte 5 of track 0 sector 1, the first sector of the super-boot program on the disk.

CHAPTER 6 - THE FORMAT PROGRAM

Once you have SK*DOS up and running, the next step is to be able to format more SK*DOS floppy disks and make backups or copies of your disk.

We provide the source code for two format programs: FORMAT is for floppy disk formatting, and HDFORMAT is for formatting the hard disk. This chapter discusses FORMAT; we leave the HDFORMAT for you to work on, since it will depend so much on the hard disk hardware you choose to use.

Formatting a brand new disk writes data into every sector of every track of the disk, numbers each sector so the FDC can find it later, and initializes the disk so that the directory is empty, the System Information Sector contains the correct information, sector 1 (and possibly sector 2) of track 0 contains the super-boot program, and the rest of the disk is one long chain of free space.

The sample FORMAT program is well documented, and not much explanation is required. It is partially table driven. The two tables, SDTABL for single density and DDTABL for double density, specify how each track will be initialized. For example,

```
SDTABL   FCB 10,10
```

specifies the number of sectors per side on track 0 (10) and the number of sectors per side on the remaining tracks (also 10). The next entry,

```
         FCB 12,$FF
```

specifies that the track entry will contain 12 bytes of $FF and so on. The last entry,

```
         FCB 1,4,7,10,3,.....
```

is the sector interleave table which specifies the physical layout of the sectors on each track; in this example, the very first physical sector is logical sector 1, followed by logical sector 4, then 7, and so on. (Interleaving is used so that logically consecutive sectors are somewhat separated on the track so that the computer is given some time between reading or writing consecutive sectors for other calculations.)

Note that the FORMAT program contains the code for the super-boot program discussed earlier.

As discussed in the User Manual, SK*DOS really does not care how many tracks or sectors a disk has, as long as there is a maximum of 256 tracks and 256 sectors per track. For floppy disks, however, we have standardized on the following numbers of sectors per track:

Disk size	Single density	Double density
3-1/2"	10	18
5-1/4"	10	18
8"	15	26

Moreover, sectors are correctly numbered on both sides (that is, sector 11, which is on side B of a single-density double-sided disk, is numbered $0B, even though it may be the first sector on the second side. This is not universal among all computers, some of which may number it $01 because it is the first sector on that side.)

Either way, the disk drivers must know which side to access for a particular sector. This is done by looking at the sector number - numbers above 10 on single density 5-1/4" disks, for example, automatically mean side B.

CHAPTER 7 - DEVICE DRIVERS

This Chapter augments the information on I/O drivers given in Chapter 14 of the Users' Manual; please read that chapter first.

Device drivers are installed by the DEVICE program, and are not meant to be directly executed by themselves. Hence they carry a .DVR extension rather than a .COM extension (and will probably need to be renamed after assembly.)

Since the driver is very tightly tied into the device selection logic of SK*DOS, it must be written in a particular way. We therefore supply several sample drivers; in writing a new driver, you should start with one of the existing drivers and modify it, rather than try to start anew. This Chapter describes the organization of the ADM-3A driver.

The driver program is divided into a header portion, and then 13 parts as follows:

The Header

This is simply the beginning of the program with some comments. Since the driver does not make use of normal SK*DOS entry points, it does not need to use the SKEQUATE library file; on the other hand, it may need a few special equates. For example, the sample ADM-3A Driver contains the equate DTBAUD EQU 51, meaning that the baud rate byte for each particular device is byte 51 of the corresponding device's entry in the device descriptor entry.

As noted in Chapter 11 of the Users' Manual, the device descriptor table DEVTAB starts at 3278(A6), or byte 3278 of the user data area pointed to by A6. The baud rate entry in the table is given as BAUDRT at 3329+80*DN(A6). For device 0 (which begins the descriptor table), this address is just 3329(A6), which works out to be byte 3329-3278 or byte 51 of that entry. Each succeeding device entry begins 80 bytes later, and in each case the baud rate byte is byte 51 of the corresponding entry. Hence the EQU statement in the device driver points to byte 51.

Part 1 - the Beginning

Part 1 of the driver contains only six bytes. The first two are a BRA instruction which is never executed, while the second two bytes are the version number. This code is only needed by the VERSION command, which can be used to check the version number of the driver. The BRA is there because VERSION looks for it.

The last two bytes contain hex $4452, the ASCII code for DR. The DEVICE driver looks for a DR in these two bytes to make sure it is installing a valid device driver.

Part 2 - Length Specification

The DEVICE installation program must know the length of a driver to determine whether it can be loaded on top of an old driver (if the new one is the same size or smaller), or whether it must be loaded in a new location (if it is longer). The LENGTH DC.L THEEND line puts in a byte which indicates the driver length from the very beginning (namely an ORG of $0000) to THEEND, an equate at the very end.

Part 3 - Entry point pointers

Eight four-byte pointers which indicate the entry points, relative to the beginning of the driver. The DEVICE program will add to these vectors the actual load address, and then put these eight vectors into the driver descriptor table.

Part 4. The actual ORG

All of the preceding data is used by the DEVICE program, but not actually loaded into memory with the driver. Hence the actual driver code is preceded by an ORG $0000 statement which redefines a new origin for the actual driver.

Part 5 - Device driver data area

This part of the driver is a data area which holds information used by DEVICE or used by the driver itself. It consists of the following:

13 bytes which contain the name of the driver. This is used only by DEVICE when it prints out the device assignments. These bytes are set as part of the driver during assembly.

1 byte which holds the device number. This tells the driver what device number it is assigned to so it can return that number if asked by a user program. This byte is initialized by the DEVICE program when it loads the driver.

4 bytes which hold the address of the corresponding device descriptor entry in DEVTAB. These bytes are initialized by the DEVICE program when it loads the driver.

1 byte which, for serial device drivers, holds the baud rate actually used. This may be different from the baud rate set in the device descriptor if a default value was used because the device descriptor baud rate was not set or was set incorrectly. This byte is set by the driver during initialization.

1 spare byte, required to align the next part of the code on an even boundary. More bytes may be added here if necessary.

Part 6 - Initialization

The next part of the code is called by DEVICE after it loads the driver, and is supposed to initialize the driver and its hardware. In the sample ADM-3A driver it also reads the desired baud rate from the device descriptor table and sets the hardware to match.

Part 7 - Input port status check

This routine is to check the input port, if any. If the port has no character ready (or if it is a purely output port) it is to return a zero condition. If a character is ready, then it should return nonzero and also return the number of characters in D5. The latter number is not used by SK*DOS at this time, but may be used in future versions with interrupt-driven drivers.

Part 8 - Input a character with echo

This routine is to input a character and also echo it to the output on the same device, if any. All 8 bits are input.

Part 9 - Input a character without echo

This routine is to input a character without echoing to the output port. All 8 bits are input.

Part 10 - ICNTRL input channel control

This routine implements ICNTRL for this device, except for ICNTRL functions 0000, 0001, and FFFx, which are handled internally by SK*DOS. In the sample ADM-3A driver, only four functions are implemented:

$0002 returns the raw 8-bit character (same as input without echo) $0003 enables function keys on the keyboard $0004 disables functions keys $0005 returns a special (decoded) character.

The sample ADM-3A driver treats the special characters as follows: When called with function 0005, the driver waits for a character from the keyboard. If the character is not an escape, or if it is an escape that is not immediately followed by another character, then it returns the character in D5 as usual, with the left-most 24 bits o D5 cleared.

But if the escape is immediately followed by another character, then the driver looks up the combination of two characters in a conversion table. If the two- character combination is in the table, then the driver returns the two characters in D6, and also returns a code in D5 which indicates the combination received. For example, in the ADM-3A the combination of escape (hex $1B) and $35 indicates function key F5. The driver will then return 00000105 in D5 and 00001B35 in D6. The former (note that bit 8 is set) indicates F5, while the latter gives the actual combination of codes received.

If the combination is not in the table, then the driver returns the error code $000001FF.

Part 11 - Output device status

This routine returns a zero if the output device is not ready to output (or if the device is an input-only device); otherwise it returns a nonzero condition.

Part 12 - Output a character

This routine actually outputs a character to the output port when ready.

Part 13 - OCNTRL output channel control

This routine implements output channel control. As for ICNTRL, codes 0000, 0001, and FFFx are handled internally by SK*DOS, so only the others need be implemented. The sample ADM-3A driver shows how a number of various codes are implemented using an output character conversion table. The sample driver uses a technique which may not work for all cases, and may have to be changed for some applications. Basically, the character table consists of pairs of words: the first word (such as $0000) is the OCNTRL code being implemented, while the second word (such as $001A) shows the character actually sent to the device (in this case $1A or control-Z). If the second word is larger than $00FF, then the number is interpreted as a pointer to a more complicated program which handles the command.

Note that this depends on (a) the terminal needing only single-character codes, and (b) the driver being large enough that all pointers are more than $FF. If this were not true, then a slightly different approach to table lookup would be needed.

Part 14 - The End

Finally, the THEEND instruction provides a label which is used in defining the length of the driver. Note that there is no transfer address (i.e., no label after the END command). This precludes the possibility of someone accidentally trying to execute the driver by giving its name as a command.

CHAPTER 8 - ADDING DATE AND TIME

A clock/calendar IC is a very useful adjunct to a computer system, and SK*DOS can make quite good use of one. This Chapter discusses three programs which deal with it.

In addition to a program to read out and set the clock, SK*DOS/68K has built- in hooks to use a clock/calendar IC for two functions:

1. To automatically enter the date when booting, so it is no longer necessary to type the date in, and

2. To substitute the time instead of a sequence number for each file in the directory.

Reading out and setting the clock is done with a program called TIME; your Configuration Disk contains the source file TIME.TXT which shows how TIME was implemented for the PTA 68K computer from Peripheral Technology Associates, which uses the Motorola MC146818 clock/calendar IC. (Incidentally, the TIME command displays the current time and date, while TIME S allows you to set the clock.)

The programs needed to read the date during booting, and to add the time to directory file entries, can be implemented in one of two ways - they can be integrated directly into the BIOS, or else can be added on later.

The sample BIOS on your Configuration Disk shows how to integrate these routines into the BIOS. As in TIME.TXT, these are written for the MC146818 Motorola clock.

The other approach, that of adding on these programs later, is illustrated in the files DATEADD.TXT and TIMEADD.TXT. These files (also for the Motorola MC146818) are intended to be assembled and then APPENDed to the main SK*DOS.SYS.

When adding these routines to BIOS, you need not worry about where to place them. But when adding them later, you need to take care. As shown, the DATEADD routine fits into some dead space in SK*DOS at $1F00-20C5. This area is part of an internal FCB used only for input redirection, and would normally not be used while the system is being booted. After booting, the DATEADD routine is no longer needed and so can be overlaid.

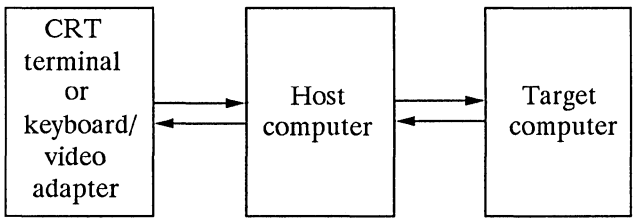
TIMEADD is needed continuously once SK*DOS is running, and so has to be placed into free memory. As the comments in TIMEADD.TXT show, you should do a LOCATE on SK*DOS.SYS to find the top memory address, then ORG TIMEADD just above that, and finally make sure to change OFFINI to point - to an even location - above the TIMEADD routine so as to protect it from user programs. The method of doing this is shown in the source code file TIMEADD.TXT.

CHAPTER 9 - OTHER MATERIAL

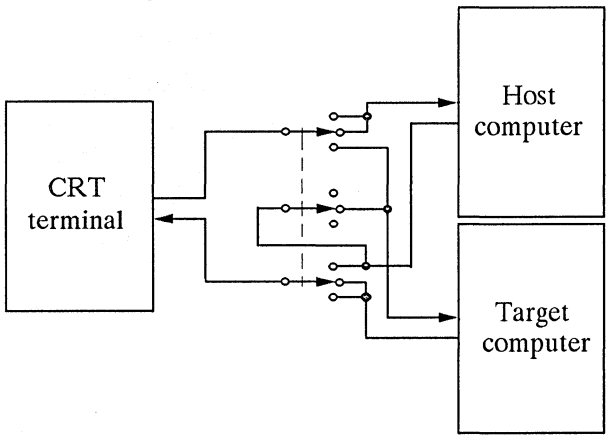
This section describes other material which you should know, and yet which doesn't seem to fit into any of the other chapters.

Sending Binary Data Over a Serial Line

The recommended method of implementing SK*DOS on a new target system is to use a host system for editing and assembly, and then to transfer binary code from the host to the target computer over a serial link. The most convenient way is to set up a connection somewhat like this:



In this configuration, the host can output binary data directly to the target computer. Furthermore, by using a communications program, the host can act as an intermediary to allow the CRT terminal to 'talk' to the target computer. But if the host computer uses a serial CRT terminal, then this arrangement requires that the host have two serial ports. An alternate method is to provide some switching as follows:



In the preceding diagram, a 3P3T switch is wired so that the three positions do the following:

Position 1. CRT terminal is connected to the host Position 2. CRT terminal sends to host, host sends to target, target sends to CRT terminal Position 3. CRT terminal is connected to the target

Position 2 is useful for downloading from the host to the target, as the CRT terminal can send the download command to the host, the host's binary output goes to the target, and the CRT terminal monitors the output of the target computer to see that all is well and no errors are generated.

There are two more-or-less-standard formats among software developers for sending binary data from one place to another - the Motorola format (which we call the S1-S9 format) and the Intel hexadecimal standard. The Motorola format is generally used by 68000 cross-assemblers for their binary output, and is also accepted by the LO command of HUMBUG. A typical line of S1-S code (usually preceded and followed by carriage return and line feed) might be as follows:

S106100089ABCDE8

The meaning of this line is as follows:

S1	Binary data follows with 2-byte address
06	6 pairs of digits follow, representing 6 bytes
1000	2-byte address where to load following data
89ABCD	3 bytes of data to load starting at 1000
E8	1's complement checksum of each byte, not including the S1 or the checksum itself

The count (06 in the above example) can range from 00 to FF, but is usually $13, representing $10 bytes, a 2-byte address, and a 1-byte checksum.

The S1 code at the beginning of the line is a marker which indicates that data follows. There are seven such markers:

S0	File name header (ignored by HUMBUG)
S1	2-byte load address is used
S2	3-byte load address is used
S3	4-byte load address is used
S7, S8, or S9	signify end of data

The S7, S8, and S9 codes often are followed by a transfer address, but HUMBUG's LO command ignores the transfer address and simply uses these codes as an end of file marker.

Since most 68000 cross-assemblers output an S1-S9 file directly, such a file can be downloaded into HUMBUG by simply listing such a file and sending its output to the target computer's serial input.

There are two other auxiliary files on the Configuration Disk, however, which convert the S1-S9 format into two other formats which might be of use.

SEND is a 6809 program which runs under 6809 SK*DOS (or Flex). Its job is to take a S1-S9 - formatted binary file and output it to the terminal port in a slightly modified way. It was originally developed for use with the ESB-I 68008 computer from Emerald Computers. We needed a way to download binary data into this system through its serial port. Although the ESB-I accepts S1-S binary data directly, it does so through a second serial port at 300 baud, and we decided we needed a faster way. The main port, which can run at 9600 baud, does not accept S1-S9 formatted data, but does accept data (presumably from a keyboard) in the format

```
MS aaaa dd dd dd dd ...
```

That is, the letters MS, followed by a load address, followed by a string of bytes separated by spaces, ending with a carriage return. So we simply wrote SEND, which outputs in exactly this format, but with a long delay after each carriage return to allow the ESB-I ROM to print its prompt. When using SEND, the ESB-I's ROM monitor receives the data as if it came from the keyboard.

It's unlikely that you will use SEND exactly as provided, but it is quite possible that you may find it useful if your 68K system does not support a S1-S9 input mode, yet does allow some other way of setting memory. Relatively simple modification of the program should be all that is needed.

SENDPROM is a variation of SEND which outputs in Intel binary format. This version was developed for sending binary data from a 6809 SK*DOS system to the Heath/Zenith EPROM programmer.

Appending Drivers to SK*DOS

Once you have BIOS written, you should save it in a binary file and append it to SK*DOS.COR. The resulting file should then be renamed to SK*DOS.SYS and LINKed so that it will be used for booting.

Assume that you have BIOS saved in a file called BIOS.BIN. (There are several ways to get it into this kind of a binary file - either assembling it with a resident assembler, or else downloading it into memory from the host computer and then using SAVE.) DIDRIV.BIN should contain a transfer address of $1000.

Making sure that there is no SK*DOS.SYS on the disk we type

```
APPEND SK*DOS.COR BIOS.BIN SK*DOS.SYS
```

(you may want to add 1. in front of file names to use a different drive) to append the two programs together into a single file called SK*DOS.SYS (note that BIOS had a transfer address of $1000, so the resulting file will also.

Next, use

```
LINK SK*DOS.SYS
```

to link it into the super-boot program. The result is a bootable disk.

Memory Usage by SK*DOS

Although the actual SK*DOS Level I code occupies only addresses from $1000 and up, and the cold-start address is $1000, some additional memory is used during booting and by the stack.

The boot ROM will load the super-boot into lower memory, (address $0800 in the sample ROM programs).

SK*DOS maintains two stacks - one just below $1000 for its own use, and another just below $0E00 for application programs and disk-resident utilities. The DOS stack will never extend down past $0E00, while the utility/application program stack will generally not go down below $0800. Hence the super-boot will usually stay safe and can be used to reboot the disk (if your system ROM has a command to jump to it.) Note, however, that the super-boot will only boot the disk it was loaded from, as it contains the starting track and sector of SK*DOS on that disk, and another boot disk is likely to have SK*DOS in a different place on the disk.

Upper/Lower Case File Names

Commands and file names entered in SK*DOS may be entered in either upper or lower case, but are automatically converted to upper case. There are times, however, when it is desired to use lower case file names or extensions (such as when using C compilers). SK*DOS does allow that.

SK*DOS contains a location called FNCASE which is used by the GETNAM routine as follows: Each character submitted for a file name is checked against the value of FNCASE; if it is larger than FNCASE, then it is converted to upper case by subtracting $20. FNCASE is initially set to $60, so that all lower case letters (whose ASCII codes begin at $61) are converted to upper case. If you change FNCASE to $7F, then lower case letters will be used without change.

NOTE

As you probably know, software and documentation are subject to constant change. The fifth revision of a program or manual is seldom the same as the first.

Even though we have revised this manual several times since its first printing, it is quite possible, even likely, that it still has omissions and errors. Even more likely, it probably has fuzzy areas which seem perfectly logical to those of us in the know, but which are totally incomprehensible to someone not already familiar with SK*DOS.

We are anxious to clear up any such problems, and invite your help. If you do spot any such omissions, errors, inconsistencies, or just plain fuzzy thinking in this manual, please let us know and we will try to correct them.

The best way is by returning this sheet as soon as you have had a chance to read and use this manual.

Thank you for your help.

Name:

Street Address:

City, State, ZIP:

Here are errors I found:

Here are areas I think you should cover better:

Here are things you forgot:

This is what I think of SK*DOS: