K-OS ONE and the SAGE
Demystifing Operating Systems
by Bill Kibler

The people at Hawthorne Technology have put together an inexpensive, but efficient operating system for 68000 computers, K-OS ONE® . The original design concept was for an inexpensive system, in which all the code was provided, so that hackers could still do something on their own. We find that most systems today have become so complex that it is impossible, in many cases, to get to the hardware directly. These companies in fact have gone out of their way to make sure that the user can not change or modify their system in any way. Now that is fine if all you want to do is run commercial packages of software.

If your desires run to making a system to protect your home, or to talk to people when you are not around, a non standard design might be more to your liking. If you are just starting to get into hardware design and want to run special programs to test out that design, multi-layers of operating system are not what you want. All these design considerations require the operating system to be simple and straight forward. The installation should be easy and provide for many options or levels of development.

All of these design considerations were behind the development of the K-OS ONE operating system. We felt that the 68K was superior to the more common CPUs in use today, but the lack of an inexpensive operating system was preventing people from discovering its features. Like any project, this one has some learning and work attached to it. Most people find operating systems a mystical concept, and feel that writing operating system programs is beyond their capbilities. What I hope to do here is demystify the topic, especially the installation of K-OS ONE.

Getting Started

The major stumbling block for most people is just deciding where to start. It took me several days of looking at various things before I could chose a direction to go. The first thing needed is a computer system. If the system is already running so much the better. If the system is not running, special problems must be handled first. What I am going to cover here is bringing up K-OS on a Sage/Stride computer. At a later time I will expand on getting a system up from scratch for the first time. What we are interested in here now is what steps are needed and what you will need to learn to get the job done.

The first place to start is learning about your current system. The Sage is a 68K based unit, mine is five years old. The unit came with all the books and software including source code for all the current PROMS AND BIOSs. Without the source code it is almost impossible to bring up older systems. It is possible with just schematics to figure out how everything talks to each other, but looking at all the older programs, will make some of the items quickly clear. I printed out all the Sage source code, about two inches worth, which is what most complex system will be—very long.

To help us understand how to start, we need to review how, and what steps, occur in getting the operating system running. The hardware on reset goes to a PROM which must contain a program to start the system. This is called a BOOT program. The boot program will initialize the system enough to start some form of operation. The better systems also contain a DEBUGGER or MONITOR, should some problem or special action be needed to bring it up. In the Sage, the PROM reads some switches on the back and determines which actions to take. Normally it will test memory, then boot the system. Options are to not test memory, and go to debugger. In the debugger, a simple command will start the system, or you can disassemble the memory.

After the reset, we have a number of functions that must be performed, such as initializing the I/O devices. The initial setting of the baud rate for your serial terminal is taken from the switch setting in the Sage. The parallel devices also need to know which lines are to be input and which to be output. The disk drives should be reset to track zero and maybe even checked as to what type they are. These are the typical actions that occur after reset. If you enter into the debugger, at this point you can explore your system or do a "IF" in the Sage which starts the booting action from a floppy drive. At that point this system goes and does its boot action which means loading a BOOTSTRAP program at a fixed location and jumping to it.

Each operating system will have its own disk format and number of files which must be loaded in order to bring up the system. Most operating systems are broken into three parts; BIOS, BDOS, and COMMAND. The BIOS stands for Basic Input Output System, and does all the talking to the hardware directly. The BDOS is your Basic Disk Operating System and provides a uniform means of having programs talk to different forms of hardware. The programs will make calls to the BDOS and it will convert them into the required number of commands needed to achieve the task requested. Typically you may have a terminal and a printer installed. By sending the proper command you can ECHO all output to your terminal to the printer. The BDOS handles the echo-ing while the BIOS actually makes separate outputs to the terminal and the printer, each being a different routine in the BIOS.

The COMMAND processor takes keyboard input and interprets it into a number of predefined operations, such as displaying a directory of the disk. To display that directory it must request the BDOS to read the disk for the directory information, format that information and then send it to the terminal port through the BIOS via the BDOS. When running programs, it is typical to replace the command processor with your program, and then reload the command processor after your program ends. That operation is called warm booting.

In the K-OS those programs are SYSTEM.BIO for the BIOS, OPERATE.BIN for the BDOS, and

COMMAND.BIN for the command processor. In the 68000 the components can talk to each other by using regular jump tables and interrupt, or trap vectors. K-OS uses both tables and vectors. To bring the system up you will need to set values for both items, but then we are getting ahead of ourselves a bit here.

Bootstrapping

We that said after reset the system can automatically boot from disk or you can do this manually. In either case the Sage process is the same, two sectors are loaded from disk into memory location 400hex and then the system jumps to it. This is typical of all boot operations, what is different is the number of sectors, location, and a special Sage signature. It is at this point that we now get out our books and determine the format of our disk. The IBM PC line of disks use a 40 track format of 512 bytes per sector and are 9 sectors per track. The PC can read, and did use, other formats, but this is now the most common format. The next bit of information we need is the location of the directory information. The directory, or the information that tells you where the files are stored, is contained in two sections, FATs and directory entries. The PCDOS system is based on the original CP/M operating system which, only had 32 bytes set aside for each entry in the directory. In CP/M, the sectors that a file used were placed with the file name, which limited it to a 16K file size before another directory entry was needed.

The PC designers wanted to add date and time, as well as to allow larger files, so their answer was using File Allocation Tables or FATs. These tables tell the operating system which sectors were used, based on a starting pointer contained in the directory entry. Typically the FATs are sectors 2 through 5 with the directory entries being sectors 6 through 12. With each side containing only 9 sectors, directory entries 10, 11, and 12 are on side 1 (the sides are 0 and 1). The bootstrap PC loader is on sector 1 only and contains data other than the bootstrap. The book I used for most of the PC information is *Peter Norton's Programmer's Guide to the IBM PC*, and I can recommend it for more details.

You need to know this information, because K-OS uses the PC disk format. Without this compatibility the porting over of the system would be considerably more complex and time consuming. All things are not totally simple however, as the Sage is not PC compatible. What we

Boot Loader Listing

```
;
;       KOSONE BOOTLOADER ROUTINE - PC COMPATIBLE
;
;
;       Boot loader routine for the Sage/Stride computer,
;         written Sept 1987 by Bill Kibler some portions
;         supplied from HTPL sample BIOS: BIOSAMPL.ASM.
;
;
;       The Sage computer loads sector 1 and 2 when given
;       a boot command "IF". Each sector is 512 bytes long.
;       The first four bytes of the boot sector must contain
;       the word "BOOT" or the boot loader in the PROM will
;       error out. The code is loaded at 0400hex and the program
;       will jump to 0404h after checking for "BOOT". Also the
;       sectors 1 and 2 are logical blocks, 0 and 1 (!). The
;       Sage PROM reads sectors by logical block numbers not
;       track, side, and sector.
;
;
;       The PC disk format is compatible as far as sector and
;       track utilization. The PC uses only the first sector for
;       boot loader with the FATs starting at sector two. PCs use
;       Clusters of two sectors to a block while the Sage is
;       one sector per block for the floppy disks. This will
;       require doubling of the cluster number and subing one
;       to get the proper sector number for passing to the
;       Sage FDREAD routines.
;
;
;       The program loads ONLY the FIRST FAT and FIRST DIR
;       sector, to save space. This means that the SYSTEM.BIO
;       or BIOS file must be loaded within the first 15 files!
;       The BIOS code is loaded at A00h, space between 400h
;       and A00h can be used after BIOS is completely loaded.


;
        TITLE   "SAGE BOOT.ASM PC COMPATIBLE BOOTSTRAP LOADER"
********************** ORIGINS ********************************
BOOT            EQU     00000004H
BOOT_CODE       EQU     00000400H           ;BOOTER LOCATION
BOOT_VAR        EQU     000005E0H           ;SCRATCH AREA
BOOT_FAT        EQU     00000600H           ;FAT READ WITH BOOT
BOOT_DIR        EQU     00000800H           ;DIR 1 LOCATION
BIOS_CODE       EQU     00000A00H           ;BIOS CODE
TERMTEXT        EQU     00FE0018H           ;PRINTOUT TEXT STRING
TERMCRLF        EQU     00FE001CH           ;PRINTOUT CRLF
FDREAD          EQU     00FE0028H           ;READ FLOPPY DISKETTE
TOTRACK         EQU     40                  ;TRACKS PER DISK
TOTSECT         EQU     9                   ;SECTORS PER DISK
TOTSIDE         EQU     2                   ;SIDES PER DISK
DEBUG           EQU     00FE0010H           ;DEBUG ENTRY
*************** SYSTEM INITIALIZATION **********************
        ORG     BOOT_CODE
;
        DC.B    "BOOT"  ; Sage checks for this statement
;
;       PROM starts program here...
;
        LEA     BOOT_VAR,A3
        MOVE.L  (A7)+,RTNADD(A3)
        MOVE.W  (A7)+,DRIVE(A3)
;
        JSR     TERMCRLF            ;flag to booting is going on
        LEA     INITMSG,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
;
        MOVE.W  #5,-(A7)            ;load first DIR sector
        LEA     BOOT_DIR,A0         ;DIR load location
        MOVE.L  A0,DIRPN(A3)        ;ALSO LOAD DIRPOINTER
        MOVE.L  A0,-(A7)            ;PUSH location on stack
        MOVEA.W #512,A0             ;one sector load
        MOVE.L  A0,-(A7)            ;PUSH sector length
        MOVE.W  DRIVE(A3),-(A7) ;PUSH drive number
        JSR     FDREAD              ;go read sector

        BNE.S   ABORT               ;go debugger if error

;
;------------------------------ LOAD BIOS

        LEA     BOOT_VAR,A3     ;setup pointer
        MOVE.L  #SYSNAM,FNAMPN(A3)      ;load string pointer
        MOVE.L  #BIOS_CODE,LOADPN(A3)   ;LOAD BIOS ADDRS
        BSR     FINDFIL
        BNE.S   ABORT               ;go debugger
        LEA     MSG1,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
        BSR     LOADFIL
        BNE.S   ABORT
```

learn here is that the Sage loads sectors 1 and 2 as the bootstrap program. Sector 2 however is the first FAT and cannot contain boot program. This leaves 512 bytes for the program, less four bytes for "BOOT". Sage not only loads the program, but then checks to see if it is the correct program. The PROM reads the first four bytes looking for "BOOT", if not found it will abort to the debugger. When found it jumps then to 404hex (just pass "BOOT") and starts the BOOTSTRAP. The bootstrap must then load the BIOS, jump to it, and then the BIOS loads BDOS and command programs.

The Real Work

The real work involves getting enough information and program samples from the Sage BOOT loader, PROM, and BIOS listings to figure out how to load the BIOS. What must also be considered is handling the FATs and DIR data as they are in INTEL hex format. The 68000 stores address or data in memory with high values followed by low values. The Intel processors store the same information in LOW then HIGH, or backwards from real life (this is one reason people like Motorola products). Not only are values in the directory stored LOW then HIGH but the FAT table has 12 bit values with the bits shifted around. It is a bit funny, so just get a book and read about it. The answers to our problem are found in the sample BIOSs supplied by Joe Bartel who wrote K-OS. These sample BIOSs show just how to manipulate the Intel bits and FATs with 68000 assembly language.

There are several ways we can boot load the BIOS. If code length was no problem, we would load all the FATs and directories, shuffle through them till we found our program, and then load them. Space being limited, I decided to cheat a bit. I let the PROM load not only the boot program, but also the first FAT. I followed that by loading the first directory sector. Next I checked those two sectors for the file and its FATs, loading same. This requires that the BIOS be loaded first, before any other programs. You can load it several times and even a few others (not more than 16), but I would experiment with a freshly formatted disk and only the three files first.

The next question is how do I get them on the disk, especially the boot loader. The PCDOS comes with DEBUG as a utility for reading disk data as well as checking memory. I would look most of the commands up in the manuals first so you understand what you are doing. This

```
;
;**************** SET UP DONE, START A PROGRAM ************************
        JMP     BIOS_CODE       ;START BIOS
;
;**************** COULD NOT LOAD SYSTEM *******************************
ABORT   LEA     MSG2,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
        JMP     DEBUG           ;EXIT TO DEBUGGER
        RTS
;
********************* SYSTEM LOAD ROUTINES *************************
;-------------------------------- FIND FILENAME IN DIRECTORY
FINDFIL
        MOVEQ   #15,D0
FIND20  MOVEQ   #10,D1
        MOVEA.L DIRPN(A3),A0
        MOVEA.L FNAMPN(A3),A1
FIND30  CMPM.B  (A0)+,(A1)+     ;COMPARE DIR ENTRY TO FILE NAME
        DBNE    D1,FIND30
        BNE.S   FIND40
        RTS                     ;RETURN TRUE IF EQUAL
FIND40  ADDI.L  #32,DIRPN(A3)
        DBRA    D0,FIND20               ;ENDFOR
        MOVEQ   #1,D0
        RTS                     ;RETURN FALSE IF NOT FOUND
;-------------------------------- LOAD BINARY FILE INTO MEMORY
LOADFIL
        MOVEA.L DIRPN(A3),A0    ;GET DIR.START
        ADDA.L  #26,A0
        BSR     LDINTELWORD
        BSR     BLKTOREC        ;CONVERT START BLOCK TO START RECORD
        MOVE.L  D0,RECORD(A3)
        MOVEA.L DIRPN(A3),A0    ;GET DIR.SIZE
        ADDA.L  #28,A0
        BSR     LDINTELLONG
        ADDI.L  #511,D0         ;CALC NUMBER OF RECORDS TO LOAD
        MOVEQ   #9,D1
        LSR.L   D1,D0
        MOVE.L  D0,RECCNT(A3)   ;FOR ALL RECORDS IN FILE
LOAD20  BSR     TRANSFORM       ;PUSH NEXT SECTOR/BLOCK

        MOVE.W  SECTOR(A3),-(A7) ;PUSH sector number
        MOVEA.L LOADPN(A3),A0
        MOVE.L  A0,-(A7)        ;PUSH location on stack
        MOVEA.W #512,A0         ;one sector load
        MOVE.L  A0,-(A7)        ;PUSH sector length
        MOVE.W  DRIVE(A3),-(A7) ;PUSH drive number
        JSR     FOREAD

        ADDI.L  #512,LOADPN(A3) ;ADVANCE POINTER
        BSR     NEXTREC         ;CALC NEXT RECORD NUMBER USING FAT
        SUBQ.L  #1,RECCNT(A3)
        BNE.S   LOAD20          ;ENDFOR
        RTS

;-------------------------------- CALC NEXT RECORD USING FAT
NEXTREC MOVE.L  RECORD(A3),D0
        BTST.L  #0,D0           ;IF ODD( RECORD ) THEN
        BEQ.S   NXRC10
        BSR     RECTOBLK        ;RECORD=BLKTOREC( NEXTBLK( RECTOBLK( RECORD)))
        BSR     NEXTBLK
        BSR     BLKTOREC
        BRA.S   NXRC20
NXRC10  ADDQ.L  #1,D0           ;ELSE RECORD = RECORD +1
NXRC20  MOVE.L  D0,RECORD(A3)
        RTS
;-------------------------------- CONVERT RECORD NUMBER TO FAT INDEX
RECTOBLK
        SUBI.L  #12,D0
        LSR.L   #1,D0
        ADDQ.L  #2,D0
        RTS
;-------------------------------- CONVERT FAT INDEX TO RECORD NUMBER
BLKTOREC
        SUBQ.L  #2,D0
        LSL.L   #1,D0
        ADDI.L  #12,D0
        RTS
;-------------------------------- GET NEXT BLOCK IN FAT CHAIN
NEXTBLK MOVE.L  D0,D1
        ADD.L   D1,D0   ;TABLEPOINTER=BLOCK*3/2+FATBUF
        ADD.L   D1,D0
        LSR.L   #1,D0
        LEA     BOOT_FAT,A0
        ADDA.L  D0,A0
        BSR     LDINTELWORD
        BTST.L  #0,D1   ;IF PREVIOUS WAS ODD
        BEQ.S   NXBL10
        LSR.L   #4,D0   ;THEN SHIFT OUT LOW NIBBLE
        RTS
```

```
NXBL10  ANDI.W  #0FFFH,D0;ELSE MASK OFF HIGH NIBBLE
        RTS
;-------------------------------- LOAD A WORD STORED IN INTEL FORMAT
LDINTELWORD
        MOVEQ   #0,D0
        MOVE.B  1(A0),D0
        LSL.L   #8,D0
        MOVE.B  (A0),D0
        RTS
;-------------------------------- LOAD A LONG STORED IN INTEL FORMAT
LDINTELLONG
        MOVEQ   #0,D0
        ADDQ.L  #4,A0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        LSL.L   #8,D0
        MOVE.B  -(A0),D0
        RTS

;--------------- CONVERT LOGICAL RECORD TO SECTOR/SAGE BLOCK
TRANSFORM
        MOVE.L  RECORD(A3),D0
        MOVE.W  D0,SECTOR(A3)
        RTS
;

**************** RUN TIME CONSTANTS **********************************
SYSNAM  DC.B    "SYSTEM  BIO",0
;
INITMSG DC.B    "HTPL-SAGE BOOTSTRAP",0
MSG1    DC.B    "LOADING BIOS",0
MSG2    DC.B    "READ ERROR",0
;
********************* VARIABLES **************************
        ORG     BOOT_VAR
;
VARS    EQU     $

;
RECORD  EQU     $-VARS
        DS.L    1       ;LOGICAL RECORD TO READ/WRITE
FNAMPN  EQU     $-VARS
        DS.L    1       ;LOAD FILENAME
LOADPN  EQU     $-VARS
        DS.L    1       ;LOAD ADDRESS
DIRPN   EQU     $-VARS
        DS.L    1       ;DIRECTORY ENTRY
RECCNT  EQU     $-VARS
        DS.L    1       ;LOAD RECORD COUNTER
RTNADD  EQU     $-VARS
        DS.L    1       ;RETRUN ADDRESS
DRIVE   EQU     $-VARS
        DS.L    1
SECTOR  EQU     $-VARS
        DS.L    1

        END
;
;     To write this to disk use the following commands
;          A>DEBUG          ;load debugger
;          -F100 1000 00    ;clear memory
;          -NBOOT.HEX       ;name of source file to load
;          -L               ;load file into memory at 100hex
;          -WCS:400 1 0 1   ;write memory starting at 400 hex
;                           ;write drive B starting with logical
;                           ;sector zero and writes 1 sector
;          -Q               ;exit to system
;
```

program can read and write data to any given sector. You can also modify files and save them by file name. To prepare the boot disk you will need to do both operations. The first step is to prepare the boot file. I used my favorite editor on the sample BIOS supplied by Hawthorne and pared it down to the essential items, and then used the calls to the PROM to load individual sectors.

There is another good reason to start with the bootloader first, it is simple, and it will show you the special considerations needed in the 68000 assembly language. Now I think the 68K is a lot easier to program than Intel chips, but the structure does require you to remember some simple principles. The 68K is a 32 bit machine and can address data either as 8, 16, or 32 bits. In assembly we use .B, .W and .L respectively for BYTE, WORD, and LONG. I got sloppy copying code from the Sage boot loader and shifted a .W for a .L. Depending on the operation this might give you the proper value, but then it might just give you all zeros instead. I kept sending those zeros until I realized what was going on.

It gets more complex when we talk about position independent code. In position independent code, all loads and stores are done off of values stored in address registers. These become base addresses and you offset or point to a memory location off of that register. Words point to the first 16 bits, with the values going to the lower 16 bits of the destination. The same operation as a Long will load the first 16 bits as high values, then the next 16 as low values. This problem became very important when calling Sage routines, as they are values pushed onto the stack. You can push ((A7)-) or pop ((A7)+) values as either words or longs, but whatever you do, both ends must be the same. I messed up and pushed some longs that should have been words, only to have unsuccessful reads.

Doing it

I have supplied some code showing what the boot loader is like, and the number of routines needed. Included with the sample is the dialog used with DEBUG to get the files on the disk. The steps go like this for the boot loader: save disk drive value (to make sure we continue to boot from it); print a message so we know we got this far; load the first DIR sector (remember the FAT was loaded with BOOT loader); find the file name and sector needed for loading the BIOS file; load those sectors; jump to the BIOS. You could save some time if you knew exactly which sectors to load, but then every time you made a minor change, the boot loader would need changing. Putting in messages may seem a luxury, but for systems that don't come up, knowing which routine failed become very important. A common way, and one possible here, is just outputting carriage returns and linefeeds. In the Sage that is a simple call or JSR to the PROM.

The Sage PROM deals with sector locations as blocks, and does not use track or side information. Their sector numbers start with ZERO and not ONE so you need to watch out for that. This made it simple as the record number, becomes the block number, which becomes the sector number and gets passed to the disk read routine. The Sage books talk about different formats, but I found that not to be true. I had forgot that the block numbers start at zero also, and had subtracted one from the record number (sectors start at 1,

Test Program

```
;
;       BIO LOADER TEST PROGRAM
;
;       USED TO SEE IF BOOTSTRAP LOADER WORKS
;       RENAME FILE TO SYSTEM.BIO FOR LOADING
;       AT 0B00H USES PROM TERMINAL I/O FOR
;       SAYING IT GOT THERE PROPERLY.....
;
BIOS_CODE       EQU     00000A00H       ;PROGRAM START
TERMTEXT        EQU     00FE0018H       ;TERMINAL STRING
TERMCRLF        EQU     00FE001CH       ;CRLF AT TERM
DEBUG           EQU     00FE0010H       ;DEBUG ENTRY
;
        ORG     BIOS_CODE
;
;
        JSR     TERMCRLF
        LEA     MSG1,A0
        JSR     TERMTEXT
        JSR     TERMCRLF
;
        JMP     DEBUG
;
MSG1    DC.B    "BIOS PROGRAM LOADED ",0
;
SCRATCH DC.L    1
;
        END
;
;   TO LOAD THIS PROGRAM USE MSDOS DEBUG AND THE FOLLOWING
;       A>DEBUG
;       -F100 3000 00   ;FILL MEMORY WITH ZEROS
;       -NBIO.HEX       ;NAME OF FILE TO LOAD
;       -L              ;LOAD FILE INTO MEMORY
;       -MA00 2000 CS:100 ;MOVE FILE STARTING AT A00 HEX
                        ;WHICH IS LOADING ADDRESS OF THE BIOS
                        ;PROGRAM, MOVING IT IN MEMORY TO 100 HEX
                        ;FOR PROPER SAVING TO DISK
;       -RCX            ;TELL SYSTEM HOW MUCH TO WRITE
;        CX: 200        ;SAVE ONE SECTOR TO DISK
;       -NSYSTEM.BIO    ;TELL NAME TO SAVE UNDER
;       -W              ;WRITE IT TO DISK
;       -Q              ;QUIT DEBUG
;   FOR THE FINAL BIOS USE RCX AND SET CS:??? TO LENGTH OF BIOS
;   TYPICALLY 1800 HEX LONG IF USING A00 TO 1FFF HEX.
;
```

records/blocks go from 0). I found that out after trying to load a simple BIOS test program (also included) and found it 200hex later in memory. This explains why any disk failures should return you to your debugger so you can check memory before a reset destroys what did happen.

The assembler I use was supplied by Hawthorne and assembles into Intel Hex format. The PC DEBUG will load those programs and you can move them around before letting it save them to disk. This assembler worked fine and only gave me problems once. I had incorrectly defined values in a table (used DS.L not DC.L) which changed the program counter which then caused all branch instructions to be out of range. That shows that it does check for programmer mistakes, which helps us rusty old dogs.

Closing

I am running a bit long, so I will try and tie up loose ends now. After the boot loader worked, I had the BIOS running (well sort of) in one day! I spent about a week studying the Sage code and K-OS samples, then a week programming the bootloader. I then took the boot loader and added terminal, printer, and a fuller disk I/O operation and used it as the BIOS. This was still making calls to the PROM and loading sectors one at a time (2 minutes to load the system), but it showed me it worked and that I was on the right track. Next I need to do the disk I/O in the BIOS with track and sector activity. I may later go back and change the BOOT loader to load the BIOS in one move, speeding that operation up. Later also I will put in interrupts and clock action, but then I will have the K-OS running and not be using the PC system.

A few fine points which need to be stressed are program locations. The boot loader in the Sage must go at 400hex. I Allocated buffer space, by putting the BIOS at A00hex. The BIOS must include or load a jump table 100hex lower than the OPERATE.BIN location. Until you have a chance to recompile the jump locations to routines into the BDOS or OPERATE.BIN, it will look for them 100hex below the starting location. Both the command and operate programs are position independent code so they can be anywhere, so could your BIOS. The only MUST do is put that jump table below OPERATE.BIN and COMMAND.BIN just above operate. I wrote and saved my BIOS as one file starting at A00hex and ending at 1FFFhex. That included the jump table and pre-zeroing out of variable memory locations (done by the assembler).

There are some other items that you must also learn about concerning the jump table. Each routine has certain items that must occur when the routine is jumped to. Typically items are pushed onto the stack (this case A4) and some status value returned on the stack after completion. Some routines must have this action, otherwise the system will go to never never land. I made lists and tables to help me out here as the manual is incomplete in this respect. I will go into more detail next time on these important steps. Till next time, read the manual and remember that the OPERATE.BIN is a HTPL program. HTPL programs must preserve registers A7 through A3 and D7. Your BIOS must not change these registers. Some variables and parameters are supplied by the BDOS as pointed to by A6. Read the HTPL user manual and pay close attention to the assembly language section.

This is by no means a complete coverage of everything needed to bring up the Sage or K-OS ONE. My major problem was choosing a direction to start with (I could have brought it up under the Sage's p-system), but once I started things fell into place easily. Next time I will cover more details about the BDOS and operating system. I will be contacting Joe about supplying more "how I did it" details as well as how he is doing on his installation manual. I am sure I missed something that you might not understand, so write us here at TCJ, and I will answer it next time. ■