NEWS

Issue No. 10
September 1990

IN THIS ISSUE

From the Editor's Desk...

Well, the summer has gone, fall and winter are almost here, and it's time to go back to computing. Hope you all had a nice summer.

We need more articles. Gordon Reeder has sent us a few, Mike Randall has one more in the queue, and we have a backlog of several more from Ron Anderson, but after that we're going to need more. I had planned to run more this month, but got carried away with information about the changes we've made to SK*DOS over the summer. In fact, I ran out of room so fast, I just kept making the type smaller and smaller until it fit! So if you suddenly decide that you need a new set of reading glasses -- join the over-40 set!

Did you hear the one about the two archeologists who meet at a convention and spend all their time bragging about their respective countries?

"My country, she is so wonderful," says the first, "we dug down 1000 meters under our capital city, and do you know what we found? *Wires!* And you know what that proves? That in my country, 1000 years ago, we had already invented the telephone!"

"That is nothing, my friend," replies the other. "We dug down 2000 meters under our glorious capital city, and do you know what we found? *Nothing!!!* What better proof could you imagine to show that, 2000 years ago in our glorious nation, we already had radio?!"

So much for the Scientific Method! Best regards from Pete Stark.

The *68 NEWS* is published and copyright © 1990 by Star-K Software Systems Corp, P. O. Box 209, Mt. Kisco NY, 10549. The editor is Peter A. Stark.

The subscription price is $10 per year. We accept display advertising at the rate of $10 per half-page (3.5" high by 4.5" wide). Readers are invited to contribute articles, letters, programs, tutorials, and other material for publication. We publish only material and advertising which, in the opinion of the editor, (a) applies to hardware or software for 68xx(x) type processors, and (b) is of a nature which would not normally be of interest to the major computer magazines. We simply do not have room for items of a very general nature, or items which pertain to very popular systems like the Macintosh or Amiga.

Please send articles or other material to us at the above address (preferably on disk); you may also fax us at (914) 241-8607, send it via modem to our BBS at (914) 241-3307, or call us at (914) 241-0287.

We thank you for your support.

SK*DOS Support for HD Drives

by Peter A. Stark

This past summer, we updated SK*DOS to support high density (HD) drives such as those used on the PC, on the PT-68K computer. The changes affect SK*DOS itself, the FORMAT utility, and also (in some cases) the HUMBUG ROMs. We have also made minor changes to DRIVE, IOSTAT, and some other utilities, but these are not essential to operation. You may get the revised programs by updating your SK*DOS disk, or by downloading them from the SK*DOS BBS at (914) 241-3307.

These new versions are primarily intended for the PT68K-4 computer, but they will also run on a PT68K-2. The primary addition is that they support a 37C65 floppy disk controller, in addition to the 1772 used until now. The 37C65 supports 1.2 meg 5-1/4 inch drives, and 1.44 meg 3-1/2 inch drives, in addition to normal 40- and 80-track drives. With a small conversion circuit, it also supports an 8 inch drive, though I don't know of too many people who will need that.

The new PT68K-4 computer can have two disk controllers — both the 1772, and also the 37C65. Because Western Digital is discontinuing the 1772, prices of this chip are expected to rise (since the only other company making it is charging over $20 for it). Hence Peripheral Tech expects that any future models of their computers will have the 37C65 only. In the meantime, though, the -4 can have both, although I suspect that most users will have only one or the other.

Although the PT68K-2 does not have a 37C65, there are a number of floppy disk controller cards available for XT systems which claim to support high density drives. I don't know whether they will ALL run with this latest SK*DOS, but the one I have here (which I got from Peripheral Tech, and which is about $40) works just fine on the -2. The only change required is that the TC line on the XT bus must be grounded, either on the motherboard or on the controller card (the TC line is the fifth line from the front of the card on the left side; ground is the first line from the front on the left side.) The TC card on the -4 is already grounded; only the -2 needs the addition.

The primary reason for modifying HUMBUG is to allow it to boot from the new controller; but the new HUMBUG also has a new IS command, which allows you to do an Initial Setup - it tells HUMBUG and SK*DOS what controller(s) you have, and what kind(s) of drives there are. The 37C65, both on the -4 computer as well as one the plug-in cards only support two floppies, so a system with both controllers can have up to six floppies on line at one time (not that anyone is likely to want to do it.)

At this time, the new controller is only supported on SK*DOS versions for the WX1 or -GEN type hard disk controllers; we figure that there are so few users with the -HDO controller that there is no need to modify that SK*DOS as well.

Disk Drives and Controllers

The SK*DOS manual has most recently been updated to cover additional features of the PT68K-4 computer. Since this computer system has different floppy disk controllers from previous versions, let's describe a few features which you should know. In particular, it is helpful to understand the kinds of floppy drives and disks that are supported. Don't let the depth of the following discussion turn you off; we give a lot more detail than you need right now, so you can come back later and appreciate the fine details.

Floppy disks come in three sizes: the very old 8" (inch) size, the newer 5-1/4" size, and the most recent 3-1/2" size. A still newer 2" size is likely to come into use in the next few years, but is not common yet.

The data on a disk is written in circular tracks. All 8" disks have 77 tracks, while all 3-1/2" disks have 80 tracks. Ancient 5-1/4" disks had 35 tracks, but modern ones have either 40 or 80 tracks.

In the 5-1/4" case, both kinds occupy slightly more than 3/4" of total width on the disk, but in one case there are 40 wide tracks, in the other there are 80 narrow tracks. By making

the tracks narrower and closer together, the 80- track drive puts extra tracks into the position that would normally be between tracks on a 40-track drive. Hence an 80-track drive can *read* a 40-track disk if it "double-steps"; that is, if it skips over the in-between areas between tracks. The 80-track drive can also *write* a 40-track disk, but a 40-track drive may not be able to read it! The reason is that the 80-track drive writes a narrower track, which leaves some unrecorded space on each side. Depending on the exact width and alignment of the track, the 40-track drive may read so much of the "garbage" between tracks that it makes too many errors to be reliable. The only way to tell is to try.

Disks also come in three densities: SD or single-density, which is now quite obsolete; DD or double-density, which is most common; and HD or high-density, which has several variations and is the newest. (What is most confusing is that some very old SK*DOS systems use single-density on their outermost track, but DD elsewhere.)

Finally, all modern floppy disks are DS or double-sided; only the very oldest drives are still SS or single-sided. (It is easy to see why we need this short introduction to the topic.)

One more complication: the tracks are divided up into smaller sections called *sectors*. In SK*DOS, each sector has 256 bytes, although other systems use 128, 512, or even more bytes per sector. The number of sectors that can be crammed into a track depends on how fast the disk turns, and on how closely the bits are crammed together in time. Most disks turn at 300 rpm (which is 5 revolutions per second), while 8" and some high density disks turn at 360 rpm (or 6 revolutions per second).

Now on to specifics. Most SK*DOS systems use a version of the Western Digital 17xx or 27xx FDC (floppy disk controller) chip. In particular, both the PT68K-2 and PT68K-4 use the WD1772 controller, which supports SD or DD disks (but not HD or high density) disks, SS or DS, and either 3-1/2" or 5-1/2" disks. These controllers could also support 8" disks, but are usually not wired up to do so.

This controller can therefore typically handle the following situations:

-----DRIVE TYPE------		----------DISK TYPE----------			
Drive Type	Tracks	Sides	Density	Sect/Trk	K Capacity
5-1/4"	40	1	SD	10	100K
"	40	1	DD	18	180 K
"	40	2	SD	10	200 K
"	40	2	DD	18	360 K
5-1/4"	80 track 80	1	SD	10	200 K
"	80	1	DD	18	360 K
"	80	2	SD	10	400 K
"	80	2	DD	18	720 K
"	40	1	SD	10	100 K *
"	40	1	DD	18	180 K *
"	40	2	SD	10	200 K *
"	40	2	DD	18	360 K *

The 1772 controller can read and write all of the above formats, but cannot format the double-stepping formats labelled with * above.

The PT68K-4, however, also has a Western Digital WD37C65 controller, which can also handle several types of high density drives of the type commonly used on PC clones. It can also easily handle an 8" drive.

The high density drives come in two types. One, commonly called a 1.2 meg HD drive, uses 5-1/4" disks, but rotates at 360 rpm instead of 300 rpm. By using 80 tracks and a data rate the same as 8" disks, it squeezes almost 1.2 megabytes of data on the disk. The second, usually called a 1.44 meg disk, uses 3-1/2" disks, but rotates at 300 rpm and uses

the same data rate as the 1.2 meg drive. Because it rotates slower, it has a capacity 20% higher than the 1.2 meg drive.

Both of these high density drives can read and write the lower density disks as well, but SK*DOS does not implement single density for them. Hence the WD37C65 controller can handle the following combinations:

-----DRIVE TYPE------		----------DISK TYPE----------			
Drive Type	Tracks	Sides	Density	Sect/Trk	K Capacity
5-1/4"	40	1	SD	10	100K
5-1/4" 40 trk	40	1	SD	10	100 K *
"	40	1	DD	18	180 K *
"	40	2	SD	10	200 K
"	40	2	DD	18	360 K
5-1/4" 80 trk	80	1	SD	10	200 K *
"	80	1	DD	18	360 K *
"	80	2	SD	10	400 K
"	80	2	DD	18	720 K
"	40	1	SD	10	100 K *
"	40	1	DD	18	180 K *
"	40	2	SD	10	200 K
"	40	2	DD	18	360 K
1.2 meg HD	80	2	DD	18	720 K
"	80	2	HD	28	1120 K
"	40	2	DD	18	360 K *
1.4 meg HD	80	2	DD	18	720 K
"	80	2	HD	34	1360 K
8"	77	2	SD	15	577.5 K
"	77	2	DD	26	1001 K
"	77	2	DD	13	500.5 K +

As before, this controller can read and write all formats listed above, but cannot format those shown with a *. (The last format, indicated with a +, is a special format using 128-byte sectors, developed for a specific customer, and not very useful for general use.)

Since the 1772 controller can only handle two kinds of drives (which really only differ by the number of tracks), the software and hardware can fairly well differentiate between the different kinds of disks without your having to tell it. The only place where you must specify what kind of disk you want is during formatting, when you must answer several questions as to the number of sides and tracks, and the density.

The 37C65 controller, on the other hand, can handle five different kinds of drives, and twenty different kinds of disks. It can generally figure out what kinds of disks you are using, as long as you tell it what kind of drives you have. You must do this in HUMBUG before you boot by using HUMBUG's IS command to do an Initial Setup.

Disk Numbers

SK*DOS users are sometimes confused by the variety of numbers used to identify the various disk drives. Disk drives are identified by three different sets of numbers:

a. Logical Drive Numbers

Logical drive numbers are what you use in normal operation when you refer to your main drive as "drive 0", or set up your RAMdisk as drive 4. The DRIVE command refers to these by "L" numbers. For example, if you have a hard drive and one floppy drive, you might set up the hard drive as drive 0 (called L0 by DRIVE), and the floppy as drive 1 (or L1).

b. Physical drive numbers

This is what the file control block calls PHYDRV, or what DRIVE calls "F" and "H" numbers. For example, issuing a DRIVE command which says

```
DRIVE L0=F1
```

tells SK*DOS that logical drive 0 should be physical floppy drive 1.

c. Drive Select numbers

The drive select number is the number that the controller thinks the drive is. For example, a floppy drive has a set of three or four jumpers, often labelled DS0 through DS2 or DS3, which identify a particular drive to the controller. Typically, the first drive on the controller is DS0, the second is DS1, and so on.

But there is a problem here - changing a drive select number requires that you move these tiny jumpers on the drive. Some manufacturers (such as IBM and Tandy) don't want their customers messing with these jumpers, and instead do the drive selection in the drive cable. Tandy used to do it by removing pins from the drive connector; IBM does it by twisting part of the cable. In IBM's case, both floppy drives are jumpered as DS1, but the cable makes one into DS0 and the other into DS1. Because IBM-style cables are so cheap, that is the approach we use with the WD37C65 controller.

As long as a computer had only one floppy controller, then the F number was the same as the drive select. Thus floppy drive F0 was DS0 on its controller. But now that we can have two controllers (as the PT68K-4 has both a WD1772 and a WD37C65) we can have two DS0 drives, one on each controller. It is HUMBUG's Initial Setup (IS) command which tells HUMBUG and SK*DOS which physical drive number corresponds to which drive select.

To properly set up a system, therefore, requires that you use the IS command in HUMBUG to tell the system which controller and drive select goes with which F number, and then use the DRIVE command in SK*DOS to tell it which F number goes with which logical drive number.

When you do the IS command, HUMBUG will ask you which controller you want to boot from; that controller will then have F0. Since the 1772 can have up to four drives, selecting it for booting will give it F0 through F3; the 37C65 would then get F4 and F5. On the other hand, if you select the 37C65 for booting, it will get F0 and F1, while the 1772 will get F2 through F5. The system will always boot from F0 when you give the FD command, so there must always be an F0 drive. Other than that, you can skip drives, so there could be an F0 and an F5 perhaps.

Any time you want to know what the relationship is between F numbers and drive select numbers, you can find out by using HUMBUG's IS command, or by using DRIVE or IOSTAT in SK*DOS.

8-Inch Floppy Disk Drives

It is unlikely that you will want to use 8-inch floppy drives, but in case you do, the WD37C65 controller can do it. But you will need a conversion cable to allow you to connect the 50-pin connector from the disk drive to the 34-pin connector on the computer. The required connections are as follows:

34-pin cable	50-pin cable	Function
2	2	Track 44-76
8	20	Index
10+16	18	Head load
12	28	Drive 1 select
18	34	Direction
20	36	Step
22	38	Write data
24	40	Write Gate
26	42	Track 0
28	44	Write protect
30	46	Read data
32	14	Side select

If you connect this adapter directly to the controller, the 8" drive will be drive select 1 on the controller; if you connect it to the twisted end of a normal floppy connector, then it will be drive select 0 on the controller.

HUMBUG Changes

HUMBUG has been changed primarily to allow it to boot from the new controller. We have added a new IS command, which asks you for the following information:

1. Which floppy controller(s) do you have. You can have either one, or both.
2. Which of the two controllers do you want to boot from. This is the controller which will be used with the FD (floppy disk boot) command.
3. For the two controllers, what kind of drives are connected. For the 1772, you can specify up to four drives, and there are three choices for each: (a) none, (b) 40-track, or

(c) 80-track. For the 37C65, you can specify up to two drives, and the choices are (a) none, (b) 40-track, or (c) 80-track, (d) 1.2 meg high density, (e) 1.4 meg high density, or (f) 8-inch.

If you specify that you want to boot from the 1772, then the 1772 drives will be numbered F0 through F3, while the 37C65 will be F4 and F5 (assuming you have the full complement). If you specify the 37C65 for booting, then the 37C65 drives will be F0 and F1, while the 1772 drives are F2 through F5.

Only the information for F0 is used by HUMBUG; the other information is stored for use by SK*DOS.

This information is stored in the battery-backed-up portion of the static RAM, so it will be preserved when power is turned off. If the information is not stored then the new HUMBUG will refuse to boot until you run the IS command to do a setup. If you use the new SK*DOS with an old HUMBUG, then SK*DOS will only be able to use the 1772 controller unless you set the data in manually. This is possible to do with the ME command, but the locations which store drive data in the SRAM are also used by the PT68K-2 HUMBUG's Basic; hence using that Basic will erase the drive data and you will have to reset it again. (The new HUMBUG avoids that problem.)

The locations which are used to store this data are:

ADDRESS	NAME	Function
FF0BE9	FL0C	Controller type for drive F0
FF0BEB	FL0D	Drive type for drive F0
FF0BED	FL1C	Controller type for drive F1
FF0BEF	FL1D	Drive type for drive F1
FF0BF1	FL2C	Controller type for drive F2
FF0BF3	FL2D	Drive type for drive F2
FF0BF5	FL3C	Controller type for drive F3
FF0BF7	FL3D	Drive type for drive F3
FF0BF9	FL4C	Controller type for drive F4
FF0BFB	FL4D	Drive type for drive F4
FF0BFD	FL5C	Controller type for drive F5
FF0BFF	FL5D	Drive type for drive F5

The twelve FL0C through FL5D bytes specify the controller and drive type for up to six floppy disk drives (a maximum of four on a 1772 controller, and two on a 37C65 controller.) FL0C through FL5C specify the controller type for the six drives normally called F0 through F5 by SK*DOS:

VALUE	CONTROLLER
00	None
17	1772
37	37C65
other	1772 (default with older versions of HUMBUG)

FL0D through FL5D specify the drive types for those six drives. These bytes are split into two 4-bit nibbles: The left four bits (representing the numbers 0 through 3 for the 1772, or 0 through 1 for the 37C65) give the physical drive number the drive is selected as, either through jumpers on the drive or through a twist in the drive cable. The right four bits specify the drive type as follows:

VALUE	DRIVE TYPE
0	None
1	360K (standard 40-track 300 rpm)
2	720K (standard 80-track 300 rpm)
3	1.1 meg (80-track 360 rpm "high-density")
4	1.4 meg (80-track 300 rpm 3-1/2" "high-density")
5	1 meg 8" (special interface cable required)

NOTE: If you have fewer drives than a controller allows, then you should number them starting with the lowest number. For example, if you have only two drives in a 1772 controller, then they must be numbered 0 and 1; neither drive can be drive 2 or 3. In particular, SK*DOS will only boot from the first drive on a controller, so there must be a drive F0.

If you have a PT68K-2 computer, but attempt to run a PT68K-4 version of SK*DOS and have trouble using your floppy disk, make sure that location FL0C (at $FF0BE9) does not have a $37 which is telling SK*DOS to use a 37C65 controller which you do not have.

Incidentally, disks which are booted on the 37C65 need no longer be linked. The purpose of linking was to tell the "super-boot" program, located on track 0 sectors 1 and 2 of a floppy, where SK*DOS.SYS is located on the disk. But since the 37C65 requires different (and longer) program code than the 1772, we decided it wasn't practical to try to squeeze a 37C65 super-boot on the disk. It wouldn't fit, and besides, it would make the disk bootable on only one kind of controller. Hence the 37C65 boot routine in the new HUMBUG bypasses the super-boot altogether, and boots directly off the disk. You still need to LINK a disk, however, if you intend to boot it on the 1772.

The Fly in the Ointment

For those of you not familiar with U.S. slang, this means "what are the disadvantages?"

As you can see the 37C65 can do almost everything the 1772 can, but there is one thing it can't do — while formatting a disk, it cannot write disk data to the disk at the same time. When the 1772 formats a disk, it simultaneously writes zeroes into each sector and *sets up the disk linkages*. So the complete format process is to format the disk, go back to the outside track, and then verify — a fairly fast operation.

The 37C65, on the other hand, first formats all tracks and writes zeroes into each sector, but (because of a basic limitation in the 37C65) it cannot write the linkages at the same time. So between the formatting and verifying, it has to go through an extra step to write the linkages. Because this is a separate operation, the 37C65 formats a disk a lot slower than the 1772. The process is particularly slow on high density disks, because there is a much larger number of sectors per track than normal.

The only reasonable way to speed up the process is to read and write an entire track at a time, rather than a sector at a time. This is a modification which we are working on right now, and we expect it to make quite a difference. It should speed up not just formatting, but other operations as well.

It will be some time before we are ready to release that change, though. The problem is to make sure that the process is safe. The problem with writing an entire track at once is that software may send data to SK*DOS to put on the disk, but SK*DOS may hold that back until it has a complete track of data before actually writing it. While this saves time, if for some reason you pull the disk out or shut off the machine before the data has been written, you will lose data. Worse yet, if you should swap disks in the meantime, SK*DOS might write the data in the wrong place on the new disk, thereby leaving the old disk incomplete, and also corrupting the new one. We're still working on alternative solutions to that problem.

Which all brings up a question several people have asked about SK*DOS, namely

Why is SK*DOS Slower than XX-DOS?

A number of users have asked why SK*DOS is not as fast as other DOSes when writing disks. To explain what is going on, let's talk a bit about how a DOS works. (In this discussion, when I use the word "DOS", I'm talking about all DOSes, not any particular one.)

The major job of a DOS is to keep track of files on a disk. This is broken down into three subjobs:

1. The DOS has to keep a list of the current files on the disk, including their names, sizes, dates, attributes, and where they are on the disk.
2. The DOS has to keep a list of free space on the disk, so it knows where to place new files.
3. When files are added to or removed from the disk, the DOS has to modify the above two lists to keep them current.

These jobs are complicated by the fact that the disk is divided into dozens or even hundreds of circular paths called *tracks*, and each track is further divided into dozens or even hundreds of *sectors*. Thus the typical SK*DOS floppy disk may have as many as 2800 sectors, while a hard disk may have 50,000 sectors or more.

When a disk is freshly formatted, new files placed on the disk occupy consecutive tracks and sectors; in other words, they are neatly stacked, one after another, beginning from the outside of the disk. But once a disk has been used a while, files are added and deleted at various spots, and the deleted files leave empty sectors between other sectors which are still used. The DOS reuses these empty sectors for new files, so as new files are written on the disk they begin to occupy these empty holes. Quite often the new files are larger than the empty holes, and so the DOS splits them into smaller sections, putting a few sectors here, a few there, the rest over there, and so on. The disk is now said to be *fragmented.*

This fragmentation affects not only existing files, but also the free space. On a well-used disk, the free space may be broken into dozens or hundreds of empty spots, scattered all over the disk. Some of these empty spots may have just one sector in them, others may consist of many sectors. You can get an idea of this by running REDOFREE on a well-used disk and looking at the track and sector display. (Incidentally, REDOFREE does not recombine these holes into larger ones unless they are adjacent to each other -- it merely rearranges the free space so all these empty holes are reused in order from outside to inside tracks, not in the order that they were freed up.)

Thus, to keep track of (1) where each file is, and (2) where all the free space is, the DOS actually has to be able to track thousands of little sectors. This raises some problems.

First of all, a file placed on a well-used disk may be broken up into many different pieces and spread all over a disk. You can even visualize the possibility of a 1000-sector file, being stored in 1000 different 1-sector chunks on the disk. This does not happen very often, but it is *possible*, and so the DOS has to be built so it can handle it. This could make the directory very complex, and so in most DOSes the directory only contains some very sketchy information about where the file is on the disk. In MS-DOS, for example, the directory only tells us where the file begins; in SK*DOS it only tells us where it begins and where it ends. Think of the "where it begins" info as being a pointer or arrow -- it is a number which points to the track and sector on the disk where the file's first sector is located.

So that raises the next question -- if the directory only tells us where the file *begins*, how do we find the rest of the file? This is done with another series of pointers, each of which points to the next part of the file. Now this is where the big difference appears between various DOSes:

1. In SK*DOS (and other *linked-chain* DOSes), these pointers are inside the file. The first two bytes of each sector of a file contain the track and sector number of the *next* sector. Once you find the first sector (from the directory listing), you follow the pointers until you get to the last sector, which has a pointer of 00-00 to signify that this is the end. The sectors are linked together to form a *linked chain* through their pointers.

2. In MS-DOS (and similar DOSes), all of these pointers are stored in a common area of the disk called a FAT or File Allocation Table. The directory tells you where the first sector is, and then you follow the pointers in the FAT to find the others.

3. In a few DOSes (such as CP/M) these pointers are stored in the directory itself. The problem here is that the directory gets very complex if a file needs more pointers than the directory entry has room for.

What about the free space? In SK*DOS, the free space is also a linked chain. The System Information Sector on track 0 lists the beginning and end of the free chain; once you find the first free sector, you follow the pointers to find the next and so on. MS-DOS and other FAT-type DOSes do not need that, because free space is signalled by having an empty pointer in the FAT table.

This helps to explain why SK*DOS is so much slower when writing a file. Before writing a file, SK*DOS has to find an empty space to write it into. To do that, it first has to follow the pointers down the free chain. For example, to write a 100-sector file, it must first read 100 pointers from 100 sectors. This tells it which 100 sectors it can use, and also which will be the next unused sector after the file; this sector will then become the first sector in the remaining free space. We call this the *pre-read* process, and you can watch it when you use COPY to copy a file to a floppy disk. COPY will display the letter "p" when it is pre-reading the empty space, the letter "w" when it is actually writing the file, and then the letter "v" when it is verifying the file to make sure it was written without errors.

MS-DOS, on the other hand, figures out where the free space is by looking for empty entries in the FAT table. In most cases the FAT table is already in memory, so it dives right in and writes the file on the disk. Note that MS-DOS does not normally verify the file after writing it, which has always struck me as odd. Considering that PC systems are paranoid to the point that they use special memory parity checking circuitry to catch very

rare memory errors, you would expect them to verify all disk writes, since errors here are much more likely. But they do not -- in MS-DOS you may use the VERIFY command to tell the DOS that you want to verify, but the normal or *default* case is no verification. SK*DOS, on the other hand, defaults to verification, though you can use the VERIFY command to turn it off. MS-DOS slows down a lot when you turn verification on; SK*DOS speeds up a lot when you turn it off. The choice is yours.

SK*DOS's pre-read is done in two different ways. In the COPY program, where we know exactly how large the file is, we pre-read as many sectors as are needed for the file all at once. Elsewhere, where the DOS does not yet know how much free space is needed, it pre-reads only one sector at a time. This is the major reason for the slow write performance of SK*DOS - we pre-read a sector, wait one whole revolution of the disk before we can write it, then wait one more revolution before we can verify it. Having to wait an entire revolution until the sector is under the head again means that it takes two revolutions of the disk for every sector written. Thus a 100-sector file takes 200 revolutions; at a rate of 5 revolutions per second for a floppy disk, this takes 40 seconds.

Now that we know *why* SK*DOS is so much slower, let's talk about the advantages or disadvantages of each approach.

First of all, 50% of SK*DOS's slowness comes from the verification. You can turn it off if you want to do a more fair comparison with MS-DOS or other systems, but I think it would be wiser to turn MS-DOS's verification ON instead. This makes things a bit slower, but it is safer in the long run.

The other 50% comes from the linked-chain structure of an SK*DOS disk. But the chain has some other advantages to compensate. One advantage comes from the ability to undelete files. Under MS-DOS, when you delete a file, all the files' pointers in the FAT table are erased. If you then decide you really want the file back, it may be very difficult to find its pieces on the disk -- even if you realize your mistake right away. Wait a few days, and you might as well give up. There are $50 and $100 programs whose main function is to recover deleted files on MS-DOS disks, and even they often fail. In SK*DOS, on the other hand, we provide an UNDELETE command which has been known to recover deleted files days and weeks later. Unless you overwrite the deleted file with new files -- which may not happen for weeks if the disk is reasonably big -- or unless you rearrange the free space with REDOFREE, the pointers are still there, and UNDELETE will bring the file back. This may not be so great if you want to cover up deleted files for security's sake, but it sure helps recover from silly mistakes.

But the linked chain approach has another advantage as well. In order to be useful, a FAT table has to be small enough so it fits into memory all at once, instead of having to be read off the disk in pieces each time it is needed; this is the only way in which it can save us time. Consider a hard disk, for example, which has 50,000 sectors. If each sector were listed in the FAT table, we would need 50,000 two-byte pointers, for a total FAT table of 100,000 bytes. With two or three hard drives, we would need 200,000 or 300,000 bytes of RAM just to hold the FAT tables! Wow!

This is obviously not practical, and so the FAT-type DOS again takes some shortcuts. First, it may use larger sectors so there are fewer of them. For example, MS-DOS uses 512-byte sectors, whereas SK*DOS uses only 256-byte sectors. Using double-sized sectors cuts the FAT table size in half.

In addition, FAT tables usually do not list individual sectors. For example, MS-DOS lumps sectors into larger groups called *clusters* (on other systems these are called *blocks* or *granules*). On a typical MS-DOS hard disk, four 512-byte sectors are grouped into a 2K cluster (and often clusters can be even larger). The FAT table then lists clusters rather than individual sectors. A disk with 25,000 double-sized sectors thus has only 6250 clusters, and therefore needs only 12,500 bytes for the FAT table pointers. This is a tremendous improvement over needing 100,000 bytes.

Although clusters save space in the FAT table, they waste space elsewhere. Suppose you save a 100-byte file on the disk. With SK*DOS, this file will use one sector. Of the 256 bytes in the sector, 100 bytes will be used and 156 bytes will be wasted. Under MS-DOS, on the other hand, this file will use a cluster. Of the 2048 bytes in the cluster,

100 bytes will be used and 1948 bytes will be wasted. That 1948 bytes could have held seven other files under SK*DOS!

On the average, the last sector of an SK*DOS file will tend to be half-empty; thus an average SK*DOS file wastes 128 bytes. An average MS-DOS file, on the other hand, wastes half of a cluster. With 2048-byte clusters, this wastes 1024 bytes -- eight times more. Put 100 files on a disk, and you will waste about 12K on an SK*DOS disk, 100K on an MS-DOS disk.

FAT-type DOSes try to reduce this loss by using different cluster sizes on different disks. Small disks will have small clusters and small FAT tables, large disks may have huge clusters and huge FAT tables. On my MS-DOS system, for example, my floppy disks use 1K clusters; one partition of my 40-megabyte hard disk uses 2K clusters; another partition on the same drive uses 4K clusters. The need for different cluster sizes and FAT table sizes makes the DOS more complex; it partially explains why sometimes changing to a newer version of the DOS requires hard disks to be reformatted, because the new DOS cannot handle the format of the old disk.

So there you have the full story. In a nutshell, a linked-chain DOS like SK*DOS is less efficient in terms of time, but more efficient in terms of space and in terms of recovering from errors. I suppose that helps to explain why the 40-megabyte hard disk on my MS-DOS system is almost full, whereas the 20-megabyte hard disk on my 68000 system, despite having tremendous amounts of source code text files for multiple versions of SK*DOS and all its utilities, has lots of room.

Beginner's Corner

by Ron Anderson

Last time we talked briefly about the 68XXX processor's Data and Address registers. We wrote a short program that used the Data registers and calls to SK*DOS routines. I'd like to start off by explaining how the SK*DOS routines are actually called or invoked. The 68XXX has some illegal instructions. All 68XXX machine code instructions are an even number of bytes long. They may be two, four or six bytes. No legal instruction starts with $A0 or $F0. The $A0xx codes cause the 68XXX to jump to an error trap routine. Peter has simply written an $A0 error trap routine that interprets the next byte as an instruction to jump to the appropriate SK*DOS routine. For those of you who have used the Motorola 6809, it is very similar to a Software Interrupt. Perhaps this is not very clear at the moment, but we have a lot of easier concepts to learn before we have to get into the detail of this. Actually you can take it at face value (it works) and not worry about it forevermore.

Well, last time we wrote a simple program to add two constant numbers and print the total in decimal and hexadecimal. If that were all we could do with computers, they wouldn't be very interesting. We learned how we could output the result of the addition last time. Now let's figure out how to ask the user to input the two numbers and then print the result. To do this simply, we will use two more SK*DOS routines, one to print a "string" and the other to input a decimal value.

```
* ADD TWO NUMBERS INPUT BY USER

*EQUATES

WARMST EQU $A01E RETURN TO SK*DOS BY JUMPING HERE
PSTRNG EQU $A035 PRINT A STRING POINTED AT BY A4
PCRLF EQU $A034 OUTPUT A CARRIAGE RETURN AND LINEFEED
OUT5D EQU $A038 OUTPUT A 16 BIT INTEGER AS A DECIMAL NUMBER
DECIN EQU $A030 INPUT A DECIMAL NUMBER FROM COMMAND LINE

START DC DECIN GET FIRST NUMBER FROM COMMAND LINE
 MOVE.W D5,D0
```

```
 DC DECIN GET SECOND
 MOVE.W D5,D4
 ADD.W D0,D4 ADD THE TWO
 LEA MSG1(PC),A4
 DC PSTRNG PRINT MESSAGE
 CLR.L D5 SET TO LEADING SPACES FOR OUT5D
 DC OUT5D OUTPUT THE RESULT
 DC WARMST BACK TO SK*DOS

MSG1 DC.B "SUM IS: ",$04

 END START
```

This program uses some features of SK*DOS that we have not used before. First, the routine DECIN, which gets a decimal number from the command line that you have typed. Then we used a literal string constant. The line DC WARMST ends the program since it causes a jump back to the SK*DOS command processor. What follows is a label MSG1 and then a "string" of characters enclosed in quotes, and lastly the byte value $04. This causes the assembler to place the message in memory starting just after the DC WARMST instruction. The label gives the assembler a "handle" on where it is. We've used a new instruction in the 68XXX set, the LEA instruction. LEA stands for Load Effective Address. The (PC) indicates that the address is to be relative to the program counter. That is, MSG1 is some number of bytes beyond the LEA instruction. The assembler adds the address of the LEA instruction to that number of bytes to arrive at the location of MSG1 and puts that memory address in Address Register A4. If you refer to the SK*DOS manual you will see that for PSTRNG to function it expects to find the address of the text string in A4, and that the last byte of the string must be 04 (04 or just 4 is the same in Hex and Decimal, so we could use $04, $4, 04 or 4 equally well for the "string terminator".

Now assemble the program using ASM ADD2 +L. Assuming you are in the A directory, run it using 1.a/ADD2 345 123 and it should report SUM IS: 468 and return to SK*DOS. Now we're getting somewhere. We have generalized our specific program to add 5 and 7 to one that can add any two numbers input by the user. Well, almost any two numbers. Remember that we have used Word length operations so that the input numbers and their sum must all be within the range -32768 to 32767, the maximum and minimum numbers that can be represented in signed binary notation with 16 bits.

Try adding a positive and a negative number (actually the same as a subtraction) and notice that the result is incorrect. DECIN, though the manual doesn't say so, apparently doesn't like a minus sign. It returns a value of 0 if you try to input a negative value.

I said last time that if there were no .L or .B on an instruction, a .W would be assumed. Generally I like to include the .W anyway for clarity and as a reminder that the length of the operand must be specified even if by default. Leaving the .W off might lead to forgetting to use a .B or a .L at a crucial place, which can lead to long debugging sessions, particularly if you are just learning to write Assembler programs. I do leave the .W off for SK*DOS calls as in DC PSTRNG. You can choose either way.

Now, the above works reasonably well, but the two numbers to be added have to be supplied on the command line. How might we ask the user for them and let him enter them? In our first program we used only Data registers. In the one above we have added a "constant" declaration, the text of the message. Now let's add space to store a "Variable". The creators of the 68XXX processors made a decision for the future users, that they should only be able to access constants (i.e. read only values) on the Program Counter Relative basis. To write to a memory location that is within the program requires some finnagling. Also, the DECIN routine in SK*DOS uses one of the SK*DOS pointers that points to the command line in order to get the value of the number there. We can still use DECIN, but we will have to fool it by altering the command line pointer. A "Pointer" is a register or a memory location that holds the Address of something else, usually some data.

This is probably moving too fast for most of you who are new to Assembler programming, but of course you have a month to digest what is here, so let's push on. There are several new concepts to discuss before we get to the next program. First we need to look at Address Registers.

```
MOVE.L A1,A0
```

moves the contents of A1 to A0. That's pretty straightforward and nothing new from what we had discussed previously except that we are using Address registers rather than Data registers. The real usefulness of them is that they may be used as pointers.

```
MOVE.B (A1),D7
```

Now that is totally different. Those parentheses around A1 change the meaning of the instruction completely. This instruction says to use the contents of A1 as the memory address from which to get a Byte value and move it to D7. This is called Register Inderect addressing. We use the value in the register as a Pointer to the data. This addressing mode has an additional feature. We can do the following:

```
MOVE.B 6(A1),D7
```

This moves the contents of the memory address 6 beyond the pointer in A1, to D7. The 6 in this case is an "offset". If an offset is not present it is assumed to be 0. This is called Register Indirect with Offset.

Let's list the program here and then discuss the remainder of the new things in it.

```
* ADD TWO NUMBERS INPUT BY USER

*EQUATES

VPOINT EQU $A000 SETS A6 TO POINT AT THE SK*DOS VARIABLES
WARMST EQU $A01E THIS TO JUMP TO SK*DOS AT END OF PROGRAM
PSTRNG EQU $A035 PRINT STRING POINTED AT BY A4
PCRLF EQU $A034  OUTPUT CR AND LF
OUT5D EQU $A038  OUTPUT DECIMAL NUMBER A4 IS POINTER
DECIN EQU $A030  GET A DECIMAL NUMBER POINTED AT BY LPOINT
GETCH EQU $A029 GET A CHARACTER FROM THE KEYBOARD
LPOINT EQU 758 VARIABLE OFFSET TO LPOINT

START DC VPOINT GET POINTER TO SK*DOS VARIABLES
 MOVE.L A6,A0 SAVE POINTER TO VARIABLES
 LEA MSG1(PC),A4 GET ADDRESS OF msg1 IN A4
 DC PSTRNG PRINT CR/LF, THEN STRING
 BSR.S GETSTR GET THE FIRST NUMBER AS ASCII CHARACTERS
 LEA STRBUF(PC),A1
 MOVE.L A1,LPOINT(A0) POINT LPOINT AT STRBUF
 DC DECIN USE DECIN TO CONVERT ASCII REPRESENTATION TO INTEGER
 MOVE.W D5,D0 SAVE IT FOR LATER
 LEA MSG2(PC),A4
 DC PSTRNG
 BSR.S GETSTR GET SECOND NUMBER
 LEA STRBUF(PC),A1
 MOVE.L A1,LPOINT(A0)
 DC DECIN TRANSLATE ASCII CHARACTERS TO INTEGER
 ADD.W D5,D0 ADD THE TWO NUMBERS
 LEA MSG3(PC),A4
 DC PSTRNG
 CLR.L D5 SET TO LEADING SPACES, NOT ZEROS
 MOVE.W D0,D4  SET UP FOR OUT5D
 DC OUT5D CALL IT
 DC WARMST RETURN TO SK*DOS
```

```
* GETSTR SUBROUTINE
GETSTR LEA STRBUF(PC),A1
GET1 DC GETCH
 CMP.B #$20,D5 COMPARE INPUT CHARACTER TO SPACE
 BEQ.S DONGET  BRANCH IF EQUAL
 CMP.B #$0D,D5 COMPARE INPUT CHARACTER TO CR
 BEQ.S DONGET  BRANCH IF EQUAL
 MOVE.B D5,(A1)+ OTHERWISE MOVE CHAR TO STRBUF
 BRA.S GET1 GO AROUND AGAIN AND GET ANOTHER CHARACTER
DONGET MOVE.B #$0D,(A1) TERMINATE ASCII NUMBER WITH CR
 RTS RETURN TO LINE AFTER bsr IN MAIN PROGRAM

MSG1 DC.B "INPUT FIRST NUMBER  ",$04
MSG2 DC.B "INPUT SECOND NUMBER ",$04
MSG3 DC.B "SUM IS: ",$04
STRBUF DS.B 30

 END START
```

In the first program we did this time, we used a literal constant, the message. This time we have three messages. The technique for using them is exactly the same. I mentioned altering a command line pointer in SK*DOS in order to use DECIN this time. See Appendix A in the SK*DOS manual for a list of SK*DOS variables that are available to user programs. You will see LPOINT listed as one of them. The SK*DOS routine VPOINT does nothing but set A6 to point at the start of these variables in memory. LPOINT has an offset of 758. In the above program we have set LPOINT EQU 758. An EQU directive causes a substitution in the remainder of the program. Wherever the word LPOINT appears, the assembler will substitute 758! Of course we could have used 758 directly, but substituting the name makes a program much more readable and more easily modified.

SK*DOS uses several of the data and address registers, but calls to it don't disturb D3-D0 and A3-A0. We did the VPOINT call and then moved A6 to A0 where the pointer to the variables will be safe.

This program introduces the concept of a subroutine. We have used a subroutine called GETSTR which gets characters from the terminal and puts them into a buffer STRBUF that we have defined at the end of the program. Notice that the constants are defined with DC.B and the buffer is defined with DS.B, which means Declare Storage bytes. DC.B allocates memory bytes, but puts specific values in them. DS.B simply sets them aside without doing anything to the data or random garbage that is in those memory locations. We have made the buffer 30 bytes long. GETSTR points A1 at the buffer with LEA STRBUF(PC),A1. It uses the SK*DOS call GETCH to get a character at a time from the terminal. CMP.B is an instruction to compare two byte values. There must be two operands following it. In this case we have CMP.B #$20,D5. GETCH places the character input by the user in D5. If the character is a space ($20) or a Carriage return ($0D), GETSTR puts a Carriage return in the buffer as the last character and returns to the main program. If the character is anything else it goes into STRBUF and A1 is incremented by 1 (That is, the pointer that shows where to put the next input character). The next instruction is BRA.S GET1. BRAnch is an unconditional instruction. Whenever program execution gets here it jumps back to the label GET1 and continues. The code from the GET1 label to the BRA.S GET1 comprise a loop. All loops must have a way out. In the present case, either of the two BRA.S instructions branch out of the loop. That is, detection of a space or CR will end the loop.

In the instruction MOVE.B D5,(a1)+. The parentheses around A1 tell us that A1 is the address of a place to move the value in D5 to. The + after the (A1) is called a post-increment operator. We moved a byte, so the post increment causes A1 to point at the next byte address in memory. If we were to move a Word, A1 would be incremented

by 2 and if a Long, A1 would be incremented by 4. This is a convenience since it allows us not to have to ADD.L #1 (or 2 or 4),A1 as the next instruction.

The BEQ.S DONGET instruction says that if the result of the comparison is equality, go to the program step with the label DONGET. The .S means "short". If the label DONGET is within 128 memory locations of the branch instruction, the .S form may be used. It generates less object code than the regular BEQ instruction which may branch anywhere in the 68XXX memory. The .S version of the branch is more efficient and faster and should be used when it is within range. Test or Compare and Branch instructions are the means by which we can control the flow of a program's execution. When control returns to the main program, It gets the address of STRBUF into LPOINT(A0) and calls DECIN. You need to look in the User's manual or a book on programming the 68XXX and study the branch instructions.

Why use a subroutine? There are several reasons. The one that seems the best one actually is not, but it is a good one. You will notice that we used GETSTR twice in our program. If it were not a subroutine we would have to include it TWICE, once at each place where it is called by the program. In some programs subroutines are called many times and it is obvious that the code will be much smaller by using a subroutine rather than repeating the required code at each place where it is needed. Actually, we have been using subroutines all along, but they have been disguised as system calls using the DC PSTRNG approach.

A better reason to use subroutines is that it breaks the program up into small "chunks" (we could say they are bite sized). A good programmer tries to break up his code into logical units that some of us call routines (particularly in discussions of assembler programs). In higher level languages they are called procedures or functions, but they amount to exactly the same thing. Rather than write the whole program at once and then start testing it, a good programmer would write the GETSTR subroutine and write a simple main program to test it. The main program would probably call GETSTR and then point A4 at it and call PSTRNG after changing the terminating $0d to $04. If the program didn't print out what you put in, you could debug that much program rather quickly. Then knowing that the subroutine works correctly, when you finished the main program you would not have to consider the subroutine if a problem were to occur. Of course if a particular subroutine were only to be used once, it could be included where it is needed and the BSR and RTS instructions dropped. It still would be reasonable to test it as a subroutine. It is also a good idea to set it apart from the rest of the code with some comment lines to indicate what it does. The name of the game is divide and conquer.

I said earlier that the Data registers and Address registers were not all there is to the 68XXX processor. The time has come to mention another, the Program Counter. The program counter is a register that always contains the address of the next instruction in the program that is running.

A subroutine makes use of one of the address registers. A7 serves a special purpose. It is used as the Stack Pointer. It is usually set to point somewhere in memory well past the end of the user program. When you do a BSR, A7 is decremented by four bytes and the address of the next instruction after the BSR is moved to the place where A7 points. The Program Counter is set to the address of the subroutine and program execution continues there. When the RTS instruction is found, the address pointed to by A7 is moved to the program counter and A7 is incremented by four bytes. All this takes place automatically. Most of the time you don't have to think about it. The operation of the stack is totally hidden when you use a subroutine. Since a subroutine can branch to another subroutine, and programs frequently do that, it is a good thing that the stack handling for BSR and RTS are automatic. Most programs use subroutines "nested" to a depth of five or six. That is, the program calls a subroutine which calls another subroutine, etc. six times.

There are times when the programmer may want to move information from the registers to the stack temporarily. See the 68000 User's Manual, and in particular the MOVEM instruction. If you don't have the Motorola manual but you do have a book on programming the 68000 in Assembler, look at the description of the MOVEM instruction

there. The most common use for MOVEM is for a subroutine to save the contents of selected Data or Address registers on the stack because it is going to use those registers for something else, and then to use MOVEM in reverse to restore the initial contents of the registers. The process is called Saving and Restoring registers. Many mysterious bugs in assembler programs are later traced to having destroyed the contents of a register in a subroutine. The cure is to save the contents and restore them at the end of the subroutine. A worse error is to save the contents of a register on the stack at the beginning of a subroutine but to forget to restore the register before the RTS command. The saved contents of the register are then interpreted as the return address for the subroutine, always with serious consequences.

Well, have we gone a long way adrift of our "Beginner's Corner" title? I don't really think so. This stuff is really fundamental to an understanding of what goes on in your computer. You now have a much better feel for what an operating system does, though we haven't begun to get into such subjects as disk files, arithmetic, etc. These fairly short and simple examples should provide some insight into how assembler programs work. If you haven't already done so, type in the last example above and assemble it. Now run it and observe how it asks you for input and calculates the sum.

One other thing. The above program is very "Fragile". It assumes that you answer the prompts correctly and doesn't check for errors. If you enter a letter rather than a number, GETSTR will accept it nicely. DECIN will cough at it, accepting it as a terminator of the number. I.e. if you put in 12A in place of 123, DECIN will see 12 as the number that you input. Sometimes a fragile program is OK if it is written to be used now and then by its author. If it is part of a system to be used by lots of others, it needs extensive error control. GETSTR in this program is used only to get integer numbers. It ought to allow a backspace to correct a bad entry. It ought to complain "Not a Number" when you input something outside of the digits 0 - 9. It should accept a CR as a terminator. Since DECIN doesn't like negative numbers, GETSTR ought to produce an error message if a minus sign is detected "Negative Numbers Not Allowed". Such refinements come later. For now, let's keep it simple so as not to obscure the point.

One final note. When you program in assembler it is very easy to write code that will hang up the computer or cause it to make an error:

```
LOOP    MOVE.B #3,D7
LOOP1   SUB.B #1,D7
        BEQ.S ESCAPE
        BRA.S LOOP
ESCAPE ......
```

The second line decrements D7 to a value of 2. The third tests for it to have reached zero, and this is the only way out of the loop. The 4th line is the error. It branches back to LOOP which puts 3 into D7 again. It ought to branch to LOOP1 which simply decrements D7 so it will reach 0 the third time through the loop.

Another very similar error is to branch to LOOP1, but to forget the SUB.B #1 instruction so that the loop count never changes.

Want to kill the computer (software wise, anyway) with two lines of code?

```
LINE1 BSR.S LINE2
LINE2 BSR.S LINE1
      END LINE1
```

This program will run briefly, the subroutines calling each other and pushing the return address on the stack until the stack fills all of memory, and then the computer will either just hang up or it will eventually print an error message. Which it will do depends on whether the stack first causes an error or overwrites the error reporting mechanism of the operating system. I just tested it and on my system it just hangs up the computer, requiring a reset to get it going again.

Well, that is much more than enough for this time.

How To Convert The Herc Driver To CGA

by Robert E. Hartge, 24 Wentz Ave., Shelby, Ohio 44875

This driver will run the HERC software on the CGA board. Thanks to Sidney Thompson for the HERC driver. The CGA.DVR and CGA.TXT files should now be available through the User's Group and Star-K BBS.

Before we start make a copy of HERC.TXT and rename it CGA.TXT. First we will explain the change we will make and why. Then we will show you the change to make.

1 - The first thing we will change is the device name. This is so the proper name will show up in the device table when the device is loaded.

```
DRNAME    DC.B      'Mono_Kbd.DVR',4
```

In the line listed above change (Mono_Kbd) to (CGA__Kbd).

2 - Next we will change the monochrome attribute table to a CGA attribute table to be used with BLINK, UNDERLINE, and REVERSE.

Find and delete the monochrome attribute table listed below. Then insert the CGA attribute table where the monochrome table was at.

```
*     VIDEO ATTRIBUTES FOR THE MONOCHROME CARD

UNDLATT   EQU    $01          Underline
BLNKATT   EQU    0            Blank
NORMATT   EQU    $07          Normal
REVATT    EQU    $70          Reverse video
NOSHOW    EQU    0
HILTATT   EQU    $08

* CGA ATTRIBUTES
REV       dc.b   $4f           reverse
NORM      dc.b   $1e           normal
BLINK     dc.b   $cf           blink
ULINE     dc.b   $5f           underline
```

NOTE You do not have to use these attributes. You may change these to whatever colors you wish.

3 - Next we have to change the values that will be written to the 6845 video controller to setup the video and sync.

Delete the top table from the monochrome text and insert the bottom table which is the CGA table. It is not necessary to put in the comments but they could be helpful if you have any sync trouble with your monitor.

```
        MONOCHROME TABLE
* work locations - necessary evils
crtint    dc.b   $61          crtc-r0 data
chrlin    dc.b   $50          crtc-r1 chars per line on crt
          dc.b   $52          crtc-r2
          dc.b   $0f          crtc-r3
          dc.b   $19          crtc-r4
          dc.b   $06          crtc-r5
lincrt    dc.b   $19          crtc-r6 lines on crt
          dc.b   $19          crtc-r7
          dc.b   $02          crtc-r8
          dc.b   $0d          crtc-r9

        CGA TABLE
* work locations
* this info. sets registers 0 to 15 in the video card 6845
```

```
crtint    dc.b  $6f          R0 horizontal total (work with sync)
chrlin    dc.b  $50          R1 number of chr.s per display line
          dc.b  $59          R2 increase to shift display left or
visa versa
          dc.b  $0f          R3 horizontal sync width
          dc.b  $23          R4 vertical total
          dc.b  $06          R5 vertical adjust
lincrt    dc.b  $19          R6 vertical lines displayed
          dc.b  $1e          R7 vertical sync position
          dc.b  $02          R8 interlace mode
          dc.b  $07          R9 scan lines per display line
```

4 - The next change is made to correct a problem encountered with carriage return and line feed at the bottom of the screen on the CGA board.

Delete the monochrome lines listed below and insert the CGA lines in place of the monochrome lines.

```
              MONOCHROME LINES to delete
ctllf2    moveq     #0,d6
          addi.w    #80,dispad(a5)          do hwd scroll
          andi.b    #$07,dispad(a5)
          move.w    dispad(a5),d6           get start disp addr
          move.l    d6,-(a7)                save for later
          addi.w    #1920,d6                addr of new line
          andi.w    #$07ff,d6
          lsl.l     #2,d6                   displace for PT68k
          movea.l   d6,a6
          addi.l    #BNWRAM,a6
          move.l    #BNWRAM+$2000,d6        end od display memaddr
          move.l    #$00200000,d7           space/attribute
          move.b    MATTR,d7                add attribute
          moveq     #9,d5
*
*  blank new line for scroll
*
ctllf4    move.l    d7,(a6)+                to disp mem
          move.l    d7,(a6)+
          move.l    d7,(a6)+
          move.l    d7,(a6)+
          move.l    d7,(a6)+                5th
          move.l    d7,(a6)+
          move.l    d7,(a6)+
          move.l    d7,(a6)+
          cmpa.l    d6,a6                   time to wrap mem?
          bmi.s     ctllf6                  br if not
          suba.w    #$2000,a6               back to logical zero
ctllf6    dbra      d5,ctllf4
          move.l    (a7)+,d6
          move.b    #13,BNWADD
          move.b    d6,BNWADD+2
          lsr.l     #8,d6                   hi byte to low pos
          move.b    #12,BNWADD
          move.b    d6,BNWADD+2
          bra       chrxit

              CGA LINES to insert
ctllf2    moveq     #0,d6
          move.w    d6,dispad(a5)           set display start to 0
```

```
        move.l   d6,-(a7)              save for later
        movea.l  d6,a6
        addi.l   #COLRAM,a6            get ram screen ram ad-
dress
        move.w   #1920,d5              set pointer for 24 lines
        move.l   #320,d6               point to chr to shift up
2 lines
ctllf4  move.l   0(a6,d6.l),(a6)+      shift 24 lines up 2 lines
        dbra     d5,ctllf4
        move.l   #$00200000,d7         space
        move.b   CATTR,d7              add attribute
        move.w    #80,d5               set pointer to blank new
line
ctllf6  move.l   d7,(a6)+              loop to blank new line
        dbra     d5,ctllf6
        move.w   #1920,linadr(a5)      set line address
        move.w   #1920,curadr(a5)      set cursor address
        andi.b   #$07,linadr(a5)
        andi.b   #$07,curadr(a5)
        move.l   (a7)+,d6
        move.b   #13,COLADD
        move.b   d6,COLADD+2
        lsr.l    #8,d6                 hi byte to low pos
        move.b   #12,COLADD
        move.b   d6,COLADD+2
        bra      chrxit
```

5 - The next corrections to be made are in BLINK, UNDERLINE, and REVERSE on and off. These corrections are needed because the CGA doesn't support these functions. We are going to have the CGA board change colors to represent these functions.

To make it easier to edit this section the monochrome and CGA text are both listed. The mono lines to delete are marked with XX as the first two chr.s on the line in the listing, and the CGA lines to insert are marked with XXX as the first three chr.s on the line in the listing. When inserting the lines omit the XXX as this is just a marker and will create problems in assembly.

```
*  Blink on
*
XX        ori.b    #$80,MATTR           blink bit on
XXX         move.b   BLINK(a5),CATTR
        bra.s    escign
*
escj    cmpi.b   #$6a,d4              'j'?
        bne.s    esck
*
*  Reverse video on
*
XX        andi.b   #$88,MATTR           kill normal display
XX        ori.b    #$70,MATTR           reverse video on
XXX        move.b   REV(a5),CATTR
*
*   kill the processing of an escape sequence
*
escign  move.w   #0,escadr(a5)        kill escape processing
        bra      chrign
*
esck    cmpi.b   #$6b,d4              'k'?
        bne.s    escl
```

```
*
*  Reverse video off
*
XX          andi.b    #$88,MATTR            kill reverse video
XX          ori.b     #$07,MATTR            normal video on
XXX          move.b    NORM(a5),CATTR
          bra.s     escign
*
escl      cmpi.b    #$6c,d4               'l'?
          bne.s     escm                  br if not
*
*  Underline on
*
XX          andi.b    #$88,MATTR            normal video off
XX          ori.b     #$01,MATTR            underline on
XXX          move.b    ULINE(a5),CATTR
          bra.s     escign
*
escm      cmpi.b    #$6d,d4               'm'?
          bne.s     escq
*
*  Underline off
*
XX          andi.b    #$88,MATTR            no reverse/no underline
XX          ori.b     #$07,MATTR             normal display
XXX          move.b    NORM(a5),CATTR
          bra.s     escign
*
escq      cmpi.b    #$71,d4               'q'?
          bne.s     badesc
*
*  Blink off
*
XX          andi.b    #$7f,MATTR            blink bit off
XXX          move.b    NORM(a5),CATTR
          bra.s     escign
badesc    move.w    #0,escadr(a5)         kill escape seq
          bra       notesc
```

6 - Next we change all monochrome references in the body of the program (not the equate table) to the corresponding CGA references. This disables the monochrome board and enables the CGA board so the information will go to the CGA board.

In the list is the monochrome reference to change from the CGA reference to change to and the number of lines that need the change. DO NOT CHANGE THE EQUATE TABLE !!!!

MONO from	CGA to	LINES
MATTR	CATTR	8
MCURV	CCURV	14
MCURH	CCURH	17
BNWRAM	COLRAM	14
BNWADD	COLADD	9

To list all the lines would take to much room so just scan through the listing after the equate table and make the changes from mono to CGA. the LINES list tells how many lines must have the mono to CGA change made to them so you may want to keep count.

At this point save the changes, and assemble the file CGA.TXT . This should generate a file named CGA.COM . Rename this file CGA.DVR. This file will be loaded by DEVICE like a serial driver or HERC.DVR .

Disk Index Program

Dan Ewers has sent me a neat little program which generates a master index of all your disks. Assuming that you have numbered your disks when formatting them (you can change numbers with DISKNAME), you then run Dan's INDEX program, which asks you to feed it all your disks in any order. It compiles a master index of all your files and which disk they are on. The master index can get saved on your system disk. You can then use INDEX to search that file for any specific program, and it will tell you where it is. INDEX can update that file to keep it current as you go on. It seems like the program can save a lot of looking. Dan has offered to send out the program to anyone interested; send $5 to cover postage and costs to Dan Ewers, L178, 12375 Military Trail, Boynton Beach FL 33436.

Disk Menu System

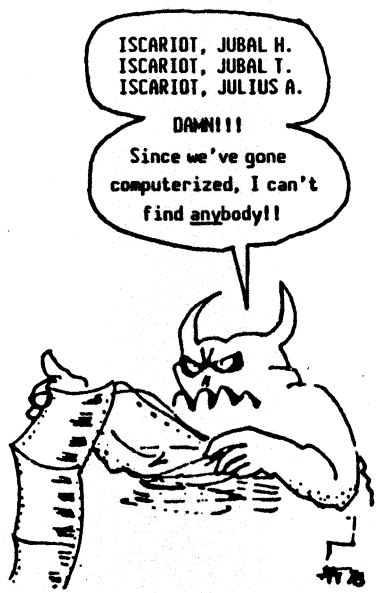
If you have seen some of the PC-DOS menuing systems such as Norton Commander, you will like Fred Stuebner's FULLLIST program.

When you start it, FULLLIST puts a listing of all your system disk files on the screen; if they don't all fit, you can scroll up or down through the list. By default, the list is sorted by date and time (latest on top), but it can also be sorted by file name, extension, or directory sequence. Using the arrow keys, you select a file by putting the cursor on it; then with just a few keypresses, you can list, edit, rename, delete, assemble, or do various other operations on the file. It makes operation of your system really easy and convenient, especially if you have a lot of files on your system.

The program is available for $30 from Fred Stuebner, 7 Kuchler Drive, LaGrangeville NY 12540, (914) 223-3336.

Applix Newsletter

For some months, we have been exchanging newsletters with the Applix users' group in Australia. The Applix is an Australian 68000 kit computer using an Australian OS called 1616/OS. For anyone interested, subscriptions to the newsletter are $10 Australian (contact your bank for the exchange rate); best payable by US money order. Order from Eric Lindsay, 6 Hillcrest Avenue, Faulconbridge, NSW 2776, Australia. I particularly enjoyed the cartoon at the right, which appeared in the Applix newsletter a few months ago.



From: Star-K Software Systems Corp.
P. O. Box 209
Mt. Kisco, NY 10549

Address Correction Requested

To: