Disk Formats and CP/M Disk Routines by M.W.T. Waters

[Ed. - this article runs to some twenty plus pages in total. I have therefore split it up, and this is the first part, concentrating on the format that data is actually written onto disks in.]

Have you ever wondered about how CP/M stores away its data, how the directory records relate to the files on disk, what determines the minimum and maximum file sizes or what determines disk size and the number of directory entries available?

There are several good books available about CP/M but none of them appear to cater for the dabbler in operating system software or for the type of 'hacker' best described as an 80-BUS user. The Digital Research handbooks contain all of the data required for a manufacturer to implement CP/M on a microcomputer but rarely explain WHY a particular disk parameter, say, is given a specific value or HOW it fits into the great scheme of things.

I should mention here that I was introduced to computers when hackers were (mainly) electronics enthusiasts who built computer systems from scratch and taught themselves programming by sheer hard work combined with more than a little trial and error. These days, hacking seems to apply to juveniles (generally) who illegally enter other peoples computer systems and create a bad name for home computer users.

Most of the information contained in this article is available to the average 80-BUS user who is armed with a disassembler, the CP/M manuals and lots of time and patience (or as someone else put it, "Stupidity and sheer bloody mindedness").

Disks and disk formats

Before proceeding with an in-depth breakdown of CP/M, I shall describe, briefly, floppy disks and floppy disk formats and then go on to examine the data physically written to the disk surface during formatting. The examples given will be oriented towards the current Gemini 5.25" disk formats but will be equally applicable to any IBM 3740 or IBM 34 type disk format (eg: Nascom).

A floppy disk consists of a disc of thin mylar (a flexible plastic) that has been coated on both sides with ferrous oxide; the same material that is used to coat magnetic recording tape. The disk surface itself is contained in and protected by a covering of some sort with cut-outs to allow access for the read/write head(s) of the disk drive. Disks are available in four sizes; 8", 5.25", 3.5" and 3" with 5.25" being the most commonly used by microcomputer manufacturers. In the case of 8" and 5.25" disks, the covering (known as the envelope) is made of cardboard while the other two sizes are protected by a rigid plastic case.

In use, the disk is rotated, usually, at a uniform speed (the Sirius microcomputer being an exception) and data is written to or read from the disk by one or two read/write heads similar to those used in tape recorders. Floppy disks may be either single or double sided. Disk drives that are designed for single sided operation only have one head while those designed for double sided operation have one head for each side of the disk. However, with early 3" disk drives, to access the second side, it is necessary to remove it from the drive and physically turn it over. For most disk operating systems (DOS's) this latter type of disk appears to be two separate disks joined back to back.

From this point on, I shall only refer to 5.25" disks but the principles involved are very similar to the other types. The disk surface is divided up into a number of tracks at the time of formatting; the number of tracks depending upon the disk size and the physical characteristics of the disk drive. The read/write head is permitted to access each track by stepping the head in or out under software control. The disk is given a reference point in terms of

rotation by an index hole which marks the start of all of the tracks on the disk. This index hole passes over an optical sensor, once every revolution of the disk, thus indicating to the disk drive that the head is about to find the beginning of the currently selected track.

Two standards have evolved for the number of tracks on a 5.25" disk. These are 48 tracks per inch (tpi) and 96 tpi although drives are now available which will achieve 192 tpi. The useful recording area of a 5.25" disk is just under a one inch band and so 40 or 80 track drives are usual although we all know of the Gemini and Superbrain 35 track 48 tpi formats, when the earliest drives had only that number of tracks. It is fairly obvious that in order to increase the number of tracks on a disk, the read/write head must write narrower strips to the disk.

Each track on the disk is divided up into a number of sectors. These sectors are generally one of 128, 256, 512 or 1024 bytes long depending upon the disk format chosen by the manufacturer. The Gemini SDDS format uses 128 byte sectors while the DDDS, QDSS and QDDS formats use 512 byte sectors. On most disk systems, the sectors are written to the disk by the format program and disks formatted in this way are known as soft sectored disks. Some systems (Apple for instance) use hard sectored disks where the sectors are physically marked on the disk by small holes similar to the index hole.

Consider, for a moment, the Gemini QDDS disk format. Disks in this format have two sides, 80 tracks per side with ten 512 byte sectors per track. The sides are numbered 0 and 1, the tracks are numbered 0 to 79 and the sectors are numbered 0 to 9. On some systems, the sector numbering starts at sector 1 so that the sectors would be numbered from 1 to 10. Some computer manufacturers start with other numbers but 0 and 1 are the most common values. This side, track and sector information is physically written to the disk during formatting so that the disk controller chip can identify the current side, track and sector by reading the disk. How this information is written to the disk will be looked at shortly but suffice to say that because the sector numbers are held on the disk as a prefix to the data held in those sectors, the disk controller can find a sector and read its data irrespective of the order of the sectors on the track. If the sectors on a track are not held in numerical order (ie: 0, 1, 2,..., 8, 9) then they are said to have been skewed.

Sector skew and its companion sector translation are used to improve access times when reading from or writing to a disk. Imagine that a disk has its sectors numbered sequentially from say 0 to 9 and that we wish to read sectors 0 and 1 in that order. Having found and read sector 0, there will probably be a delay while the processor is deciding that the next sector it wants is sector 1. Meanwhile the disk will still be turning and by the time that a request is sent to the disk controller to read sector 1, that sector will have probably passed under the disk head and the controller will have to wait until sector 1 comes around again. To overcome this problem, some manufacturers including Gemini allow the disks sectors to be physically skewed during formatting. If we look at a track on the disk, the sector numbers may look like this:

0 7 4 1 8 5 2 9 6 3

In the example given, if we now read sector 0, two sectors will be allowed to pass under the disk head before sector 1 comes round and the processor should now have plenty of time to make its mind up. Obviously, the amount of skew employed depends very much on the speed of the processor and too much skew is as bad as too little when it comes to slowing down disk access.

Sector translation uses a similar principle to sector skew except that it is a software measure to achieve the same result. With sector translation, a table of sector numbers is held in the computer memory. This table may look like that given below:

0 3 6 9 2 5 8 1 4 7

What happens now is that although the physical sectors will be in sequential order, the data is read from or written to the sector pointed to by the

translation table (known as logical sectors). In other words, using the table, when sector 0 is required we find that the data will be read from sector 0 but when sector 1 is required, we actually read from sector 3. If you compare the results of the skew table and the translation table you will see that they have an identical effect with one exception. A disk that has been written using sector translation must be read using the same translation table, otherwise rubbish will result as the sectors will be read in the wrong order. If, however, a skewed disk is read on a machine that would normally use a different skew factor (ie: perhaps the skew was set up for a 2MHz processor and we are reading on a 4MHz processor), the only penalty would be one of speed of access. It is worth noting perhaps that Gemini skew the system track of the disk differently from the data tracks. On the system track, the sectors are recorded in the order shown below while the sectors on the data tracks are skewed by 0, 1, 2 or 3 as chosen by the user. The example sector skew shown a little earlier had a skew factor of 2.

Example of sector skew on the system track:

0 2 4 6 8 1 3 5 7 9

Getting back to the physical disk format, let us now look at what is physically written to the disk during formatting. Most disk controllers format the disk by writing a complete track at a time. To do this, the host microcomputer will have assembled a memory image of the track which it will then transfer to the FDC (floppy disk controller) chip. The data consists of gaps, index and address marks, track, side and sector numbers, CRC bytes and of course the areas for saving the data.

Let us break the track up into its component parts and examine them in detail starting with the track, side and sector information and the area for the data.

Each disk sector is preceded by an identification block containing information about that sector. This block is six bytes in length and is identified to the FDC chip by an ID address mark immediately before the block. The data in the block is as follows:

1 byte - Track Number
1 byte - Side Number
1 byte - Sector Number
1 byte - Sector Length
2 bytes - CRC bytes

The track and sector numbers on the disk may lie in the range 0 to 255 although this will obviously be limited to the number of tracks that the disk drive is capable of accessing and the number of sectors that will fit on one track of the disk. The side number takes the value 0 or 1. The sector length byte (with the FD1797) may fall in the range 0 to 3 and the values correspond to sector sizes of 128, 256, 512 or 1024 bytes respectively. The two CRC bytes are automatically computed by the FDC chip and will be written to the disk when instructed to do so by the host computer.

When the FDC is instructed to read a sector, it first waits for an ID address mark and then reads the ID block. If the side, track and sector information matches the values given by the host computer, and if no CRC error has occurred, the FDC will transfer data from the data area following the ID block. The data area will be of the length indicated by the sector length byte and is preceded by a Data Address mark on the disk. Initially, the data area contains the value 0E5H for every byte. This value is set during formatting but in fact any value could have been used. 0E5H is used, by convention, because IBM used this value in their original floppy disk formats (back in the dim and distant past when 8" floppies were used with mainframe computers and micros had not yet been invented). Finally, the data area is terminated with 2 CRC bytes for error checking.

Writing a sector is similar to reading but after the correct ID block has been found, a data address mark is written to the disk followed by the number of data

bytes indicated by the sector length byte. Finally, the two CRC bytes are written to the disk followed by one byte of ones (0FFH).

In the Gemini DDDS, QDSS and QDDS formats, there are ten sectors per track and consequently the pattern of ID blocks and data areas will be repeated ten times on the track but with differing sector numbers.

Normally, the memory image of the track to be written during formatting would look something like that given below; the values given conform to the IBM System 34 format. When using the FD1797 FDC, the index mark is not required and so Gemini have left it out together with the pre-index gap (gap 4) and the 0F6H bytes. In the example memory images given, the centre column contains the data sent to the FDC while the right hand column shows the data written to the disk surface.

IBM System 34 Format

	Number of bytes	Hex value of byte sent to FDC	to disk	
	80	4E	4E	
	12	00	00	
	3	F6	C2	
	1	FC	FC	; Index mark
	50	4E	4E	
*	12	00	00	
*	3	F5	A1	; Resets CRC generator
*	1	FE	FE	; ID Address mark
*	1	Track No	Track No	
*	1	Side No	Side No	
*	1	Sector No	Sector No	
*	1	01	01	; Sector Length (256 bytes)
*	1	F7	CRC1	
*			CRC2	
*	22	4E	4E	
*	12	00	00	
*	3	F5	A1	; Resets CRC generator
*	1	FB	FB	; Data Address mark
*	256	E5	E5	; Empty Data area
*	1	F7	CRC1	
*			CRC2	
*	54	4E	4E	
**	598	4E	4E	

* Write this field 26 times.
** Continue writing 4EH until next index pulse received (Physical end of track).

By contrast, the track format for the Gemini QDDS format is given below:

Gemini QDDS Format

	Number of bytes	Hex value of byte sent to FDC	to disk	
	32	4E	4E	
*	12	00	00	
*	3	F5	A1	; Resets CRC generator
*	1	FE	FE	; ID Address mark
*	1	Track No	Track No	
*	1	Side No	Side No	

```
*      1          Sector No     Sector No
*      1            02            02       ; Sector Length (512 bytes)
*      1            F7          CRC1
*                               CRC2
*     22            4E            4E
*     12            00            00
*      3            F5            A1       ; Resets CRC generator
*      1            FB            FB       ; Data Address mark
*    512            E5            E5       ; Empty Data area
*      1            F7          CRC1
*                               CRC2
*     30            4E            4E

**   512            4E            4E
```

* Write this field 10 times.
** Continue writing 4E until next index pulse received (Physical end of track).

As can be seen from the examples, there are gaps containing zero and/or 4EH bytes between the ID blocks and data areas, before and after the index mark (if present) and at the end of the track. These gaps are there to allow the FDC to synchronize with the disk to read the data and ID blocks.

The gaps before the index mark and at the end of the track are known as Gap 4. The gap after the index mark and before the first ID block is known as Gap 1. Gap 2 separates each ID block from its associated data area while Gap 3 separates the data area from the next ID block on the disk.

The values of the bytes used to create the gaps are different for single density and double density modes as are the number of bytes used. Below is an extract from the manufacturers data sheet for the FD1797 showing the byte values and byte counts for the gaps. The values shown for the byte counts are minimum except where shown.

	Single Density	Double Density
Gap 1	16 bytes FF	32 bytes 4E
Gap 2	11 bytes FF	22 bytes 4E
*	6 bytes 00	12 bytes 00 3 bytes A1
Gap 3	10 bytes FF	24 bytes 4E
**	4 bytes 00	8 bytes 00 3 bytes A1
Gap 4	16 bytes FF	16 bytes 4E

* Byte counts must be exact.
** Byte counts are minimum except exactly 3 bytes of A1 must be written.

The physical data is held on the disk surface as a serial data stream. In double density mode a 250nS pulse is sent to the disk drive for each flux transition. This would imply that a pulse is sent to the drive each time the data changed from 0 to 1 or 1 to 0. In addition to the data, a clock is recorded on the disk surface. This clock is picked off when reading the disk to synchronize the data being read. The presence of a clock on the disk is used to good effect by the FD1797 when sending control bytes to the disk.

As seen in the example track formats, byte values of 0F5H and 0F6H generate the values 0A1H and 0C2H on the disk surface. However, so that these bytes can be distinguished from data bytes of the same values, clock pulses are deliberately

missed out by the FDC. In fact, when sending the 0A1H byte, there is no clock pulse sent between bits 4 and 5. Similarly, when sending the 0C2H byte, the clock pulse between bits 3 and 4 is missed.

In single density, various other clock pulses are missed when sending control bytes to the disk. In the table below, when sending a data byte, if all 8 clock pulses associated with that byte are present then the clock can be considered as having the value 0FFH. A missing clock in any position may be represented by a zero bit in the clock byte. The table also shows the values written to the disk for the values sent to the FDC.

Byte sent	Single Density	Double Density
00 to F4	Write 00 to F4 with Clk=FF	Write 00 to F4
F5	Not allowed	* Write A1, preset CRC
F6	Not allowed	** Write C2
F7	Generate 2 CRC bytes	Generate 2 CRC bytes
F8 to FB	Write F8 to FB, Clk=C7, preset CRC	Write F8 to FB
FC	Write FC with Clk=D7	Write FC
FD	Write FD with Clk=FF	Write FD
FE	Write FE, Clk=C7, preset CRC	Write FE
FF	Write FF with Clk=FF	Write FF

* Missing clock transition between bits 4 and 5.
** Missing clock transition between bits 3 and 4.

File sizes, disk sizes and directories

Having well and truly taken a disk to pieces, we can at last return to CP/M and the questions asked at the beginning of this article. (Can anyone remember what they were?) Well, you'll have to wait until the next episode.

A Look at MultiNet 2 by P.A. Greenhalgh

MultiNet Design Philosophy

The aim of Gemini's MultiNet networking system is to provide computing facilities to a number of people for the minimum possible cost. As a very significant proportion of the cost of any system is in the mass storage and hard copy devices, the overall cost of a multiple system installation can be dramatically reduced by allowing a number of users to share these facilities. 'Share' in this instance means the ability for any user to be able to make use of the mass storage device, but not, in general, the sharing of the stored data.

CP/M Compatibility

As Gemini MultiBoard systems are all capable of running the CP/M operating system, and given the amount of applications software available for that operating system, MultiNet is designed to provide a CP/M 'compatible' environment, the major difference being that the user need not have physical disk drives present at his Workstation, but the MultiNet software refers all disk requests to a Fileserver.

To achieve this, software has to be written that looks to the applications program as though it is CP/M, but in reality this software contains no disk driver or file handling routines, but instead refers these to the Fileserver that is controlling the mass storage. To achieve this the relevant CP/M documentation is used to write software that meets the given specifications as closely as possible, given the major premise that there are no physical drives present. Unfortunately, in practice, it is found that certain programs make use of certain 'quirks' or 'undocumented features' of CP/M, and so the emulation software has to be modified in order to provide as identical an environment as possible. It is thus extremely difficult, if not impossible, to achieve 100 per cent compatibility.

Disk Formats and CP/M Disk routines - Part 2 by M.W.T. Waters

Part 2.

File sizes, disk sizes and directories

Having well and truly taken a disk to pieces in Scorpio News V1 I1, we can at last return to CP/M and the questions asked at the beginning of the first part of this article. (Can anyone remember what they were?)

Minimum file size on disk is determined by the block size (referred to by Digital Research as BLS). CP/M stores files on disk as one or more blocks of information, where each block may contain 8, 16, 32, 64 or 128 CP/M records (128 bytes long) depending upon the block size. Blocks may be 1K, 2K, 4K, 8K or 16K bytes long, chosen (usually) by the computer manufacturer when deciding upon disk size and format. As an example, there are four commonly used Gemini formats available. The SDDS format uses a 1K block size, the DDDS format uses a 2K block size while the QDDS and QDSS formats use a 4K block size. The blocks are sometimes referred to as allocation units since disk space is allocated to files as one or more blocks.

The minimum size of a CP/M file is one block. If you imagine that you wish to save a short utility program of, say, 128 bytes to disk, that program will appear to be 1K long on a system with a 1K block size or 16K long with a 16K block size. On a system with a 16K block size, there would be 16256 bytes of disk space wasted after the file was written to disk. Why then do manufacturers, such as Gemini, use larger block sizes? At first sight, it would appear that a 1K block size is the best choice all round. As we shall see later, the problem of which block size to use isn't quite that simple otherwise all manufacturers would have used 1K blocks (wouldn't they?).

What then, determines the maximum size of a file or disk? To quote Digital Research, CP/M 1.4 allows a maximum disk size and file size of 256Kbytes. The maximum disk size may be extended by the use of double density but the process is cumbersome and wasteful of directory space. For this reason, few computer manufacturers appear to have made double density CP/M 1.4 systems.

CP/M 2.2 allows a maximum disk size and file size of 8Mbytes. The reason for this is that a maximum of 65536 records of 128 bytes each may be written to a disk or file. The maximum file size is determined by the maximum value for the random record field of a CP/M 2.2 file control block which may range from 0 to 65535.

Under CP/M 3 (CP/M Plus), the maximum disk size allowed is 512Mbytes made up of a maximum of 32768 blocks. This of course assumes a 16K block size; smaller block sizes reduce the maximum disk size accordingly so that with a 1K block size, the theoretical maximum disk size is 32Mbytes. In practice, the maximum number of 1K blocks under CP/M 2.2 or CP/M 3 is 256. This is to provide compatibility with CP/M 1.4 disks as we shall see later. The maximum file size allowed by CP/M 3 is 32Mbytes and is determined again by the maximum value for the random record field of the CP/M 3 file control block which may range from 0 to 262143 giving 262144 records of 128 bytes each per file. To accommodate this higher figure, the lower 2 bits of the R2 byte of the FCB are now used to give an 18 bit random record number.

At this point, we need to see how directory information is stored on a disk. For those familiar with DU.COM, this may appear to be revision but don't skip this section too quickly as a couple of surprises may be in store for you.

A CP/M 2.2 directory entry on disk looks something like the example given below. The format used to show the entry is "borrowed" from DU.COM.

```
00544553 5446494C 45434F4D 00000001   *.TESTFILECOM....*
2D000000 00000000 00000000 00000000   *-................*
```

The characters shown between the asterisks are the ASCII representation of the Hex characters on the left. A full stop is used to show unprintable characters.

The format for the directory entry corresponds almost exactly to that of a CP/M file control block and for good reason. Apart from the first byte, the directory entry is a direct copy of the first 32 bytes of the FCB.

The disk directory information is stored on the data tracks of the disk (as opposed to the system tracks) and occupies the first one or two (usually) blocks on the disk. Each directory entry requires 32 bytes and so it follows that for a 1K block size, each block will hold 32 directory entries. If you should wish to double the number of directory entries available to 64, then 2 blocks will have to be set aside for the directory. Therefore, the number of directory entries available and consequently the maximum number of files that may be stored on a disk is a trade off against the size of the disk and the number of blocks that can be spared for directory information. Gemini opted for 128 directory entries per disk with a 4K block size, so only one block is lost to the directory leaving 196 blocks for the files themselves.

Looking at the entry for TESTFILE.COM, the first byte tells us that the file is stored in user area 0. The user number is held in the lower four bits of the byte. The upper four bits are reserved by Digital Research and although they are unused in CP/M versions up to 2.2, CP/M 3 does use them so, if anyone was thinking of having 256 user areas, think again if you wish to remain compatible with future releases of CP/M.

A value of 0E5H in this position indicates to the BDOS that the file has been erased. This makes utilities such as UNERASE possible as, to restore a file, all that is necessary is to change this byte to a value between 0 and 15 depending upon which user area is required to hold the file. Unfortunately, this may not always work as if a file has been written to the disk since erasure, some of the data blocks used in the erased file may have been reallocated to the new file. For users of CP/M 1.4, it is not possible to unerase a file as the ERA command completely wipes the directory entries for the erased file by filling them with 0E5H bytes.

CP/M 3 uses different values of the first byte of the directory entry to identify disk labels, password information and date/time stamps, all of which occupy directory space and consequently reduce the number of files that may be stored on the disk. The disk label occupies one directory entry and may be identified by a 20H value for the first byte. Date/Time stamps and passwords are invoked by formatting the directory using the utility INITDIR.COM which reserves one directory entry for every three file entries in the directory; reserved entries being identified by the value 21H in their first byte. You will see that if Date/Time stamping is invoked under CP/M 3, the maximum number of files that may be stored on the disk is immediately reduced by one quarter.

Next in the directory entry comes the 11 bytes for the file name and type. The last four bytes in the top row give file size information and will be looked at later. Lastly, the 16 bytes in the bottom row contain the block numbers that have been allocated to the file. CP/M gets clever here and the block numbers are stored differently depending on whether the disk has more than 255 or less than 256 blocks available. If there are less than 256 blocks on the disk, CP/M stores the block numbers as 8 bit values and can consequently hold information on 16 blocks. If, however, the disk has 256 or more blocks, the numbers are stored as 16 bit values (stored in Low-High order) with the result that only 8 blocks may be allocated per directory entry. In the example given we can see that only one block has been allocated to the file although it isn't apparent from the entry whether 8 or 16 bit numbers are being used. The decision as to whether 8 or 16 bit numbers are used is made by the CP/M BDOS and is totally transparent to the user but the manufacturers of the Tatung Einstein force their implementation of CP/M to use 16 bit values, even though their disks have less than 256 blocks. This, presumably, was an attempt to make their disks unreadable on other systems (Didn't fool Dave Hunt though).

A quick tap on the calculator keyboard will have told you that, assuming say, a 2K block size and 8 bit directory entries will allow a maximum of 32K to be referenced by a directory entry. This implies (correctly) that more than one directory entry will be required to hold the block numbers for larger files. CP/M automatically takes care of this problem for us. When writing to disk, if a directory entry is filled, CP/M closes it (by writing it to disk) and opens a new one. It should be noted that each time CP/M has read or written all of the blocks allocated by a single directory entry, it has to read the next entry from, or write the next entry to disk. Logically, it would appear that to improve the performance of the disk system (i.e. speed up access), the block size should be as large as possible so that CP/M would need to access the directory track of the disk less often (since extra time is needed to move the disk drive head from where it is now to the directory track and back again). It would also appear that with a 16K block size and 8 bit directory entries, CP/M would only have to access the directory track once for every 256K of a file that it is reading or writing. However, maybe due to a hangover from CP/M 1.4 (which always held 16Ks worth of block information per directory entry), CP/M 2.2 and CP/M Plus STILL access the directory track every 16K.

If we return briefly to the number of blocks reserved for directory entries, we can see that we must have enough to allow access to the available disk space. With the Gemini QDDS format, each directory entry can refer to 64K since 8 bit entries are used with a 4K block size. The minimum number of directory entries required to access the 196 data blocks on disk will be 13. However, if we were to take the average sized file to be about 8K then to fill a disk with them we shall need at least 98 directory entries. In practice, Gemini chose 128 directory entries which, while entirely adequate for most purposes, try filling a disk with 4K files or programs. When all 128 directory entries have been used, there is still 272K of free space remaining. To be fair, this is an extreme example and will occur very rarely (unless you use dbaseII, in which case it will probably occur daily as the average dbaseII .CMD file seems to fit in a 4K block).

The choices, then, facing a manufacturer when deciding upon block size are a compromise of disk size, number of directory entries and speed of access. If Gemini had used a smaller block size for their QDDS format (2K for example), CP/M 2.2 (or CP/M 3) would have used 16 bit entries in the directory. Consequently, four times as many directory entries would have been required. Why four? Each directory entry would refer to 2K blocks - hence twice as many directory entries would be required to hold the block numbers for the same sized file BUT each block number would now occupy twice as much room in the directory entry so double the number of directory entries again. Similarly, the BDOS would have to access the directory track four times as often with a corresponding reduction in disk performance.

A further complication that prevents the use of a 1K block size with larger disks is due to the way CP/M itself works. Let's have a quick history lesson.

CP/M 1.4 uses single density disks (although some micro manufacturers have made it work with double density) with a fixed 1K block size and 8 bit directory entries. Each directory entry, therefore, refers to 16K (which we all recognise as an extent). CP/M 1.4 allows a file/disk to contain a maximum of 16 extents numbered 0-15, hence a maximum file/disk size of 256K (16 x 16K). A hangover from CP/M 1.4 is that each directory entry MUST be able to control at least one extent. If we are using 16 bit directory entries, each block must be at least 2K long since the directory can only hold 8 block numbers. This applies equally to CP/M 2.2 and CP/M 3.

We have already seen that the directory occupies at least one block in the data area of the disk. Under CP/M, the directory blocks are always the first ones in the data area and it follows that block 0 will always contain directory information. This being the case, at no time will CP/M ever need to allocate block 0 to a file. For these reasons, zeros in the directory entry may be used to signify that no further blocks have been allocated to a file.

This may have been stating the obvious as I had already said, when referring to the example directory entry, that only one block had been allocated to the file. However, the concept of using zeros to show unallocated blocks is fundamental to the way directory entries relating to random files are interpreted. I shall return to random files shortly.

Let's finish looking at the example directory entry by examining the last four bytes of the top row. The first of the four bytes is the extent byte. This shows the highest extent number accessed by the directory entry. In our example, the highest extent accessed is 0. Had the directory entry been full on a QDDS disk, the extent number would have been 3 since this directory entry would control extents 0-3. A subsequent entry would have controlled extents 4-7 and would have 7 in the extent byte if the entry was full. Under CP/M 1.4, as already stated, the highest extent number allowed is 15 (16 extents numbered 0-15) giving a maximum file size of 256K. Under CP/M 2.2 and CP/M 3 the maximum value allowed for the extent byte is 31 (32 extents) giving a maximum file size of 512K.....Hang on a bit! Digital Research told us that the maximum file sizes for CP/M 2.2 and CP/M 3 are 8Mbytes and 32Mbytes respectively.

Under CP/M 2.2 or CP/M 3, when a file exceeds 512K, the S2 byte is used to hold the multiples of 512K and the extent byte starts numbering again from zero. This use of the S2 byte is not at all well documented. In fact I have searched high and low through the Digital Research manuals for references to the S2 byte and its use in this context with little joy. The result of this lack of information has led to the production of programs, both commercial (Wordstar) and Public Domain (D.COM, SD.COM and DU.COM) that cannot handle files larger than 512K.

The maximum value for the S2 byte under CP/M 2.2 is 15 (16 permutations numbered from 0 to 15) and 16 x 512K is 8Mbytes made up of 65536 records of 128 bytes each. Under CP/M 3 its maximum value is 63 (64 permutations) and of course 64 x 512k is 32Mbytes made up of 262144 records of 128 bytes each. Under CP/M 1.4 the S2 byte is unused in this context. The S1 byte is unused by all versions of CP/M up to 2.2 and is probably not used by CP/M 3, at least I haven't yet found where it uses it, if in fact it does.

The last byte of the four is the record count byte and it shows how many records are stored in the last extent written (i.e. the one indicated by the extent byte). If this byte has the value 80H then the extent is full. In our example, one record has been written to extent 0. Had the extent number been 3, for example, then extents 0-2 would be assumed to be full. However, this isn't necessarily true when referring to random files as we shall see now.

With the block size chosen and having established that we are using 8 bit directory entries, as we have already seen, each directory entry controls up to 64K of a file. In fact the directory entry is more than a place to store block numbers, it is literally a map of the file.

Directory entries for random files

We can split the 64K controlled by the entry up into 512 CP/M records of 128 bytes with these records being numbered from 0 to 511. Each of the sixteen 4K block positions will contain 32 CP/M records and therefore the first block will contain records 0 to 31, the second 32 to 63 and so on. If only record number 32 is written to file TEST.RND, its directory entry will look like the one given below with only the second block position containing a block number.

```
00544553 54202020 20524E44 00000021   *.TEST    RND....*
002D0000 00000000 00000000 00000000   *.-............*
```

If, on the other hand, only the 512th record had been written (record number 511), only the last block position would contain a block number.

The RC, EXT and S2 bytes will be set up as if the file had been written sequentially, i.e. writing the last record to an extent will cause the RC byte to read 80H. The only way that CP/M knows whether a block has been allocated or

not is by whether there is a zero in the block position of the directory entry. What CP/M doesn't know is which records have been written to the block; this is left for the programmer to determine.

One way of doing this is to use the write random with zero fill function when writing to a previously unallocated block. To establish, from within a program, whether or not a block has been allocated to a file is to read the desired record using the read random function. If an error code is returned, you are trying to read a block that has not yet been allocated, i.e. CP/M found a zero in the relevant position in the directory entry. If this happens, you can write the record with zero fill. If no error code is returned, a block has already been allocated which could contain the wanted record. If you previously used the zero fill function when writing a record to this block, examination of the contents of the required record will reveal zeros if nothing has been written to it or data if something has been written.

A point worth noting is that the CP/M DIR command will only show a file if a directory entry exists for extent 0. For this reason, writing one record to record position 65535, for example, will result in two directory entries being created; an empty one covering extent 0 and one covering the 128th record of extent 512.

Another observation about random files is the way that PIP.COM handles them (or more accurately, it doesn't). PIP.COM reads and writes files sequentially and will therefore get an end of file code from the BDOS when it encounters an unallocated block in a file. Normally, this would be adequate as most files indeed end this way. If faced with the example of a file that contains only record 65535, PIP.COM will only copy the empty directory entry for extent 0.

BIOS data structures

Some questions that still need to be answered are: how does CP/M know how many blocks the disk contains, what size are they, how many of them are reserved for the directory and how does it keep track of which ones are free for use? The short answer is that the BIOS contains this information but to answer these questions in detail, some information about BIOS data structures is required.

The disk routines in the BIOS contain data areas and buffer space that will be used by the BDOS when it needs to have information that is system dependent. These areas are known as the Disk Parameter Headers (DPH's), Disk Parameter Blocks (DPB's), Checksum Vectors and Allocation Vectors. We'll look at these in the next issue.

MAP 80 Multi-Purpose Interface (MPI) Review by P.D. Coker

The user of 80-BUS compatible products has been quite well served for floppy disk controllers - initially by the Gemini GM809, then the GM829 with SASI support for a Winchester, closely followed by the even more supportive GM849 - Map 80's VFC was also capable of controlling 5.25" drives and Nascom/Lucas also produced an FDC card. So why have Map produced yet another card?

The MPI is more than just an FDC, catering for the usual range of disk sizes, with support for a Winchester drive or two. By changing a few links and the FDC chip one can convert to Nascom FDC compatibility rather than the more usual Gemini/Map "standard". It also has a CTC and two channels of SIO, one of which is a standard RS232 and the other conforms to the proposed standard for RS485 (high speed multi-drop interface). The CTC can be used to generate software selectable baud rates or, if the user wishes, on-board crystals can be used to generate two frequencies on the MPI board which are independent of the system clock. It uses 16 ports and, in my experience, can substitute for either of the two earlier Gemini FDCs - I have no experience of the most recent Gemini FDC.

Disk Formats and CP/M Disk routines - Part 3 by M.W.T. Waters

Disk Parameter Headers

The CP/M 2.2 DPH

A DPH for CP/M 2.2 is a table of eight 16 bit words broken up as follows:

```
DPH:    XLT     ; Translate table address
        0000    ; 6 bytes scratchpad for BDOS use
        0000
        0000
        DIRBUF  ; Address of a buffer used
                ; for reading directory information
        DPB     ; Disk Parameter Block Address
        CSV     ; Checksum Vector address
        ALV     ; Allocation Vector address
```

Note that CP/M 1.4 doesn't use DPH's and that the DPH's for CP/M 3 are longer.

The BIOS has a DPH for each drive supported by the system. When a drive is logged in, the BIOS returns the address of the appropriate DPH to the BDOS. The BDOS then uses the DPH to locate any remaining data structures that it requires.

The first entry is the address of a sector translate table which the BIOS will use to convert from logical to physical sector numbers before accessing the disk. If no sector translation is required, this word will contain zero. Sector translation is discussed in some detail elsewhere so all that will be said at this point is that CP/M 2.2 only allows for translation of 128 byte sectors. If sector translation is required on systems using sector sizes of more than 128 bytes, XLT must be set to zero and the BIOS should do the translation separately. CP/M 3 doesn't suffer from this limitation as all disk transfers and sector translation are done in terms of physical disk sectors.

The next 6 bytes are reserved for use by the BDOS as workspace. Their initial value is immaterial.

DIRBUF is the address of a buffer used by the BDOS and BIOS when reading directory information. This buffer is one physical sector long (512 bytes for Gemini DDDS, QDDS and QDSS formats, 128 bytes for SDDS format) and may be shared by all drives in the system. CP/M 1.4 is different in that it uses the default buffer at address 80H for this purpose. CP/M 3 buffers the directory records differently and consequently CP/M 3 DPH's do not contain this field.

DPB is the address of a Disk Parameter Block for the drive concerned. A separate DPB is required for each disk format handled. DPH's for drives using the same disk format may all address the same DPB. The DPB is explained in full later.

The Checksum Vector

CSV is the address of a table that CP/M uses to check whether a disk has been changed. The table is used to store a checksum byte that uniquely identifies a directory record (128 bytes containing 4 directory entries). A good definition of a checksum is, in this case, a single byte which is derived from a block of data and whose value is unique to that block. As far as we are concerned, a block is 4 directory entries totalling 128 bytes (i.e. one CP/M record) and the checksum byte is obtained by adding together all of the bytes in the record, ignoring overflow, to produce a total. This total, and the totals for the remaining directory records are stored in a part of the BIOS known as the checksum vector and its address is given as CSV in the DPH. There is a checksum vector for each drive supported by the system. The length of each vector depends upon the number of directory entries. Since the checksum applies

to a directory record containing four 32 byte directory entries, the total space required for each checksum vector is (DRM+1)/4 where DRM is one less than the total number of directory entries permitted on the disk. In the case of the QDDS format, the figure is 128/4=32 bytes.

Whenever the directory of a disk is accessed, a checksum is produced and compared with the appropriate value in the checksum vector. If the checksum is different, the BDOS knows that the disk has been changed and will set it R/O under CP/M 1.4 or CP/M 2.2 or log in the new disk under CP/M 3.

Clearly, this directory checking is a waste of time for drives, such as a Winchester hard disk, where the disk cannot be changed. This type of drive is said to have fixed media. Digital Research had the foresight, when designing CP/M 2.2 and CP/M 3, to allow for this type of drive. If no directory checking is required, the checksum vector may be omitted with a resultant saving of space in the BIOS.

The Allocation Vector

The last entry in the DPH is the allocation address. This is the address of a table which the BDOS uses to keep track of the data blocks on disk. This table is called the allocation vector and it is a bit map of the blocks on the disk such that each individual bit represents one block. Consequently, the state of eight disk data blocks (ie allocated to a file or not) can be shown in one byte of the allocation vector. Block 0 is represented by bit 7 of the first byte in the table, block 1 by bit 6 and so on for each byte in the table. There is an allocation vector for each disk drive in the system and the address of the vector for the logged in drive may be obtained by calling BDOS function 27.

When a drive is logged in for the first time, the BDOS scans the directory entries on the disk and extracts the numbers of the blocks already allocated to files. For each block used, the BDOS sets the appropriate bit in the allocation vector. Quite simply, when you wish to write to disk, the BDOS searches the allocation vector for a 0 bit. If none are found then the disk is obviously full. If a 0 bit is found then the BDOS may allocate the associated block to the file.

To compute the amount of free space on a disk, it is a simple matter to count the number of zeros in the allocation vector and then multiply by the block size. Programs such as STAT.COM obtain their free space figures in this manner and as you will probably noticed do not access the disk at all when, for example, STAT B: is typed in while currently logged into drive A and assuming that drive B has been accessed since a ^C was last typed.

The BDOS updates the allocation vector during disk write and delete operations so that the amount of free space shown by STAT.COM will always be correct. Having said that, imagine that we are writing to a disk. The BDOS will set bits in the allocation vector according to which blocks are being allocated. If we return to CP/M by executing a RET instruction so that a warm boot is avoided and, at the same time neglect to close the file, STAT.COM will return an incorrect amount of free space as the allocation vector will not agree with the disk directory entries actually written to the disk. This is because the final directory entry for a file being written to disk isn't written to the directory track until the file is closed. The only way to reclaim the lost disk space and obtain a correct free space figure is to force a disk reset by typing Control C.

Under CP/M 1.4 and CP/M 2.2, if a disk is changed without performing a warm boot to reset the disk system, the amount of free space indicated by STAT.COM will be inaccurate as the BDOS will have the allocation vector set up for the previous disk. Under CP/M 3, as soon as the BDOS realizes that a disk has been changed, it logs in the new disk and sets up the allocation vector accordingly.

The length of the vector is calculated to be (Number of blocks)/8 bytes long, rounded up to the nearest whole byte. In the Digital Research manuals, this is quoted as (DSM/8)+1 where DSM is one less than the total number of blocks contained on the disk. Here is another good reason for having larger block sizes in that the bigger the block size, the fewer blocks there will be and the smaller the allocation vector will be (i.e. the BIOS will be smaller and the TPA, consequently, may be larger).

The CP/M 3 DPH

By contrast, a CP/M 3 DPH is given below. Digital Research needed to increase the amount of data provided by the DPH while providing upwards compatibility with CP/M 1.4 and CP/M 2.2. Further, they have come to realise that many systems require to access disks of different formats and that few systems use a 128 byte sector size on their disks.

```
        READ    ; Address of sector read routine
        WRITE   ; Address of sector write routine
        LOGIN   ; Address of disk login routine
        INIT    ; Address of disk initialisation routine
        UNIT    ; FDC relative drive code for this drive (Byte)
        TYPE    ; BIOS scratchpad. Current density etc. (Byte)
XDPH:   XLT     ; Translate table address
        0000    ; 9 bytes scratchpad
        0000
        0000
        0000
        00
        MF      ; Media Flag (Byte)
        DPB     ; Disk Parameter Block address
        CSV     ; Checksum Vector address
        ALV     ; Allocation Vector address
        DIRBCB  ; Directory Buffer Control Block address
        DTABCB  ; Data Buffer Control Block address
        HASH    ; Hash Table address
        HBANK   ; Hash Table Memory Bank (Byte)
```

The major differences between a CP/M 2.2 and CP/M 3 DPH are quite obvious at a glance. The DPB entry is as described in the section concerning the CP/M 2.2 DPH and will not be repeated here. Under CP/M 3, all disk transfers are in terms of physical disk sectors and the sector translation table, at long last, now also applies to physical sectors of whatever size. This means that it is no longer necessary to provide special translation routines in the BIOS for sectors that are larger than 128 bytes. Note also that the BDOS scratchpad area has been increased to 9 bytes and that there is no entry for DIRBUF.

Additionally, the DPH for CP/M 3 has been extended and is now known as the eXtended Disk Parameter Header or XDPH. For compatibility with CP/M 2.2, the BIOS still returns the address of the DPH when a drive is logged in and in fact returns the address of the XLT field of the DPH as with CP/M 2.2.

Before examining the XDPH fields in detail, a word or two is required about the CP/M 3 Drive Table.

The drive table (@dtbl in the CP/M 3 manuals) is a 32 byte area of RAM that is comprised of 16 addresses. Each entry in the table points to the XDPH for its associated drive so that the first entry applies to drive A and the last entry applies to drive P. If no physical drive exists, the table entry is zero but there must be an entry of some sort for each of the 16 drives that may be supported. The BDOS can obtain the address of the drive table by calling BIOS function 22 (ie calling address WBOOT+3FH in the BIOS jump table) but for most purposes, the BIOS is the main user of the table.

When a drive is logged in by calling the SELDSK entry in the BIOS jump table, the BIOS uses the drive code supplied by the BDOS as an index into the drive table and it then extracts the address of the XDPH for the required drive. If this address is zero then no physical drive exists and an appropriate error code is returned to the BDOS. If the drive does exist, the drive is logged in and the drive code is stored away for reference by the disk read and write routines (in variable @cdrv).

Similarly, when the READ or WRITE entry points in the BIOS jump table are called, these routines use the stored code for the current drive to index into the drive table and fetch the address of the appropriate XDPH. Bear in mind that any errors due to there being no physical drive present should have already been found by the SELDSK routine.

Once armed with the address of the correct XDPH, the SELDSK, READ and WRITE routines use it to extract any other information they require. A detailed description of each field of the XDPH is given below.

The INIT field provides the address of an initialisation routine for use with the drive addressed by the XDPH. The drives are initialised by the BIOS during cold boot. Other than this, the INIT routine will not be used except, perhaps, during error recovery.

The READ and WRITE fields of the XDPH are fairly self explanatory. These fields provide the addresses of the sector read and write routines for the selected drive and are extracted from the XDPH by the BIOS when called at the appropriate place in its jump table. Before either of these routines is called, other calls to the BIOS will have performed any necessary sector translation, logged in the appropriate drive, set the track and sector numbers, set the DMA address and indicated which memory bank is to be used in the coming disk transfer. All that is required of the BIOS in its simplest form is to select the required memory bank and transfer data to or from one physical disk sector.

A facility exists within CP/M 3 for the BDOS to instruct the BIOS to transfer multiple sequential sectors to a contiguous area of RAM, starting at the DMA address. The BDOS tells the BIOS how many sectors to read or write and the BIOS will, if this facility has been implemented, continue transferring sectors to or from disk until the required number have been copied. In case this facility has not been implemented by the writer of the BIOS, the BDOS will continue to set track and sector numbers and DMA address and will call the BIOS the appropriate number of times. It is up to the BIOS to ignore the surplus calls if it has already transferred the required amount of data.

The LOGIN field is used by SELDSK to find the address of the routine appropriate to the disk drive being selected.

The UNIT field is a single byte that holds the FDC relative drive code, i.e. the actual value that is output to the disk controller to select the required drive for use. The BIOS extracts this code when any of the SELDSK, READ or WRITE routines is called.

The TYPE entry of the XDPH is a scratchpad location for use by the BIOS. It may be used for any purpose but Digital Research recommend that it is used to hold drive relevant information such as current density in systems that support single and double density.

The CSV entry under CP/M 3 is identical to that of CP/M 2.2 but the checksums contained in the vector are calculated in a slightly different manner. Instead of adding together all 128 bytes in a directory record, CP/M 3 produces sub-totals for each of the four 32 byte directory entries in the record. These four sub-totals are then exclusive ORed together to produce the checksum byte. This XORing together reduces the number of bits lost due to overflow (due to adding as for CP/M 2.2) and therefore each checksum byte contains more information about the directory record. For this reason, the likelihood of identical checksums being returned for directory records on different disks is reduced and hence the possibility of a disk change being missed is also reduced.

ALV is similar to that for CP/M 2.2 in that the BDOS uses the allocation vector to identify which data blocks have been allocated to files and which are free for use. CP/M 3, however, uses two bits in the allocation vector for each data block on the disk. The only exception to this is in a non-banked CP/M 3 system where the option is given to use single bit entries as with CP/M 2.2. Since there should be no non-banked CP/M 3 systems sold commercially (the non-banked version of CP/M 3 is merely an aid in the generation of a banked system), all users of CP/M 3 can assume that their system uses double bit entries in the allocation vector.

Obviously, a double bit allocation vector is going to occupy twice as much memory as a single bit allocation vector so why then use two bits per entry? Remember how STAT.COM calculates free disk space under CP/M 2.2 and how it is possible to fool it by forgetting to close a file. The only way to update the allocation vector under CP/M 2.2 is to hit ^C to reset the disk system and log in the drives again. This of course takes time and CP/M 3 boasts better disk performance. Digital Research have gone to great lengths to avoid resetting the disk system with each warm boot - particularly since CP/M 3 has also spent time buffering both directory and data blocks to reduce disk accesses still further. Control C is still available under CP/M 3, when running the CCP, to allow the user to force a disk reset but, to improve performance, this should be done as little as possible.

To eliminate the need for disk resets on warm boot, Digital Research came up with the double bit allocation vector. If, as before, a file is written to disk, the BDOS will keep track of which data blocks it is allocating by using one of the two bits in the allocation vector for each of those blocks. When a file is closed, the second set of bits for the affected blocks are updated to match the first. If, on the other hand, the file is not closed before returning to CP/M, the first set of bits can be reset to their original state because the second set provide a record of which blocks remain free. Consequently, the disk free space shown by SHOW.COM (CP/M 3's equivalent to STAT.COM) is always correct and the need for a disk access has been alleviated. Unlike CP/M 2.2, the size of the allocation vector for CP/M 3 is calculated as (DSM/4)+2. The "+2" gives us the clue that the double bit allocation vector is in fact configured as two single bit allocation vectors joined end to end.

MF is referred to as the media flag. This byte is set to zero by the BDOS when a drive is logged in. If the disk/computer hardware supports "door open" interrupts, the BIOS can set this byte to 0FFH when a drive door is opened. If this is the case, before the BDOS next performs a file operation on the affected drive, it will check for a disk change and perform a login if necessary.

NOTE. The Media Flag is used in conjunction with the variable @MEDIA contained in the CP/M 3 System Control Block and will normally only be implemented on systems supporting a "door open" interrupt.

DIRBCB is the address of the directory buffer control block on non-banked CP/M 3 systems. On banked systems, DIRBCB points to the head of a list of buffer control blocks (BCB's). The list head is a 16 bit address which points to the first BCB in the list. The first and subsequent BCB's contain a 16 bit address field which points to the next BCB in the list while the last BCB contains a zero value in this field. This type of list is, for obvious reasons, known as a linked list. A comprehensive description of the BCB format is given later so, all I shall say at this point is that each BCB contains the address of a directory buffer which will be one physical sector in length (sound familiar to CP/M 2.2 users?). The maximum number of directory buffers required will be (DRM+1)*32/Physical sector size. For the Gemini QDDS format, this figure will be 8 per drive. Different drives may share directory buffers under CP/M 3 and consequently one or more drive XDPH's may refer to the same BCB list.

DTABCB is the address of the data buffer control block on non-banked CP/M 3 systems or a list head on banked systems. The BCB format is identical to that for directory BCB's except that each data BCB points to a data buffer which will be one physical sector in length. CP/M 3 uses the physical record buffers for blocking and deblocking the physical sector into 128 byte CP/M records and

consequently doesn't require buffers where the physical sector size is 128 bytes (Blocking and Deblocking are discussed later). When used, CP/M 3 discards these buffers and overwrites the data in them on a least recently used (LRU) basis so that when access is required to a recently used sector, CP/M 3 reads the data from memory rather than from the disk giving an appropriate increase in performance. CP/M 3 doesn't use the buffers at all if it knows that one or more complete physical sectors are to be read and hence deblocking of the data will not be required. In this case, the BIOS is instructed to read the disk data directly to the TPA at the DMA address.

CP/M 3 directory hashing

The HASH parameter contains the address of a table used for directory hashing. Hashing is a term used to refer to the process of converting a string of characters into a single number that is unique to that string. There are a vast number of methods of doing this, all varying in complexity. The problems encountered with hashing are guaranteeing that the number produced is in fact unique to the particular string involved (and no other) while keeping the number small enough to be manageable.

The problem of uniqueness can be explained with the use of an example. A simple way of converting a character string into a number is to add the ASCII values of the characters together. Taking combinations of the characters A-D for simplicity, there are 16 permutations of those characters that will give the same result. If we then permit characters to repeat, the number of combinations giving identical results increases still further. A few examples are given below, values are in decimal:

```
"ABCD" = 65 + 66 + 67 + 68 = 266
"BDAC" = 66 + 68 + 65 + 67 = 266
"AADD" = 65 + 65 + 68 + 68 = 266
"ACCC" = 65 + 67 + 67 + 67 = 266
```

One solution to this problem is to weight the character values differently depending upon which position they occupy in the string. If we were to assign values to the characters such that A=1, B=2 etc., and multiply the character values by multiples of 27 depending upon their position, the same strings as we used above would give different results as shown below:

```
"ABCD" = 1 + (2 * 27) + (3 * 54) + (4 * 81) = 541
"BDAC" = 2 + (4 * 27) + (1 * 54) + (3 * 81) = 407
"AADD" = 1 + (1 * 27) + (4 * 54) + (4 * 81) = 568
"ACCC" = 1 + (3 * 27) + (3 * 54) + (3 * 81) = 487
```

This example is fine as far as it goes but we want to hash a directory entry of 11 characters. Using the method described above, the lowest number produced will be 1486 for a file "AAAAAAAA.AAA" and the highest number will be 28636 for file "ZZZZZZZZ.ZZZ". However, filenames can also contain figures, spaces and some other characters in addition to upper case letters. CP/M 3 needs to include the user area byte and both extent bytes (EXT and S2) so that it can differentiate between files with the same name in different user areas and also between different extents of the same file. All of this serves to increase the range of numbers generated as a result of hashing.

In CP/M 3, Digital Research hash the file name, user area and extent bytes into a four byte number which constitutes the hash table entry for the directory entry. The user area number is stored as the 4 least significant bits (lsbs) of the first byte. The file name is reduced to an 18 bit number of which the lower 16 bits are stored in the two middle bytes of the hash table entry and the 2 most significant bits (msbs) are stored in the 2 msbs of the first byte. The extent and S2 bytes are combined to form a directory entry number for the file such that the first entry number is 0 and directory entry number 'n' is numbered n-1. This entry number is truncated to form a 9 bit value and is stored so that the msb is held as the 6th bit (bit 5) of the first byte of the hash table entry and the lower 8 bits are stored in the last byte. The method of hashing used on

the file name is fairly complex and involves adding the characters into an 18 bit partial result. Characters whose ASCII values are odd numbers are simply added in while even ones are right shifted first. The position of the character in the file name determines whether the partial result is left shifted none, one or two places after the addition.

The directory entry number for the file is calculated by dividing the absolute extent number by EXM+1 where EXM is the extent mask as defined in the disk parameter block. The absolute extent number is formed by joining the S2 and EXT bytes end to end to form an 11 bit number. (Under CP/M 3, there are 6 bits for the S2 byte and 5 bits for the EXT byte.) This number represents the highest extent number addressed by the current directory entry and falls in the range 0 to 2048 (2048 * 16K = 32Mbytes).

Considering the Gemini QDDS format for a moment, we know that each directory entry can control 4 extents. If we take the absolute extent number and divide this by 4, we will have calculated the directory entry number for the current entry.

All this is fine but why hash the directory entries of a disk at all? CP/M 3 maintains a table of hashed directory entries for each drive in the system. Each table is (DRM+1)*4 bytes long allowing four bytes per directory entry as previously stated. When CP/M 3 needs to access a directory entry for a file, it is a relatively simple matter to hash the file control block and search the hash table for a matching entry. Once this entry is found, its position in the hash table tells CP/M 3 precisely which physical disk sector contains the entry and enables it to directly access the wanted sector rather than having to sequentially read the disk directory from the start. Of course, if the wanted physical sector is already in a physical record buffer, this makes the process faster still.

HBANK refers to the memory bank containing the hash table for the associated drive.

That's all again for this issue. The concluding part of this article will be in the next issue of Scorpio News.

Dealer Profile - Off Records

[Ed. - In our continuing series looking at the roots of various 80-BUS dealers, this issue it is the turn of Battersea based OFF Records.]

By the year 1975 computing on mainframes had become quite tedious. A single machine had to support so many users that it was impossible to do the more interesting experiments on them and computing centres had become bureaucratic empires which stifled all work other than the trivial and routine. People who have only worked on personal computers find it difficult to visualize the straight jacket imposed by a large time-shared machine. They are fine if you happen to be a physicist running your application programs, but if you are actually interested in the machine itself, the lack of access to the innards of the machine becomes quite frustrating.

The mathematics department at South Bank Polytechnic was fortunate in having acquired three years earlier the first GT11 in the country, an early version of a PDP11 with graphics capability. For most of the time this machine could be used as a personal computer, and a very nice computer it was too! Gradually, however, more and more basic teaching had to be done on it and inevitably the machine became time-shared losing all of its early advantages. The powers-that-be also had the curious notion that, once you had bought a computer, there was no need to maintain or enhance it to cope with changing requirements. Even such a simple device as a floppy disk drive could not be added to enable students to look after their own data.

Disk Formats and CP/M Disk Routines - Part 4 by M.W.T. Waters

This is the final part in this series. But first of all here is a correction to the last part, part 3, published in Volume 1, Issue 3, page 11:

In the article, the author had a brainstorm and said that the directory buffer (DIRBUF) under CP/M 2.2 is one physical sector long. This, of course, is quite incorrect. The length of this buffer is one logical record (128 bytes) long. The description of the directory buffers controlled by the Buffer Control Blocks under CP/M Plus IS correct, however. CP/M Plus, as stated in the article, controls buffers, each of which is one physical sector long.

CP/M 3 Buffer Control Block format

The format of a Buffer Control Block is given below. The format is identical for both directory and data BCB's.

```
DRV       ;  8 bits - Source drive of buffered data
REC       ; 24 bits - Record Position on disk
WFLG      ;  8 bits - Flag indicating unwritten data
00        ;  8 bits - BDOS scratchpad
TRACK     ; 16 bits - Track number for buffered data
SECTOR    ; 16 bits - Sector number for buffered data
BUFFAD    ; 16 bits - Address of buffer for this BCB
BANK      ;  8 bits - Bank no of buffer for this BCB
LINK      ; 16 bits - Address of next BCB in list
```

DRV indicates the source drive from which the contents of the buffer located at BUFFAD was obtained or, alternatively, the destination drive for the data in the buffer.

REC contains the record position of the current buffer contents. This figure is an absolute physical sector number relative to the first sector in the data tracks of the disk. For example, the first sector in block 0 (i.e. the first directory sector) is numbered zero, the next is number one and so on for all data blocks on the disk.

WFLG is set by the BDOS when the buffer associated with the BCB contained new data that has yet to be written to the disk. When the data has been written, this flag is then set to zero by the BDOS.

TRACK is the physical track location of the contents of the buffer.

SECTOR contains the physical sector location of the contents of the buffer.

BUFFAD contains a 16 bit address showing the location of the physical sector buffer associated with this BCB.

BANK contains the memory bank number holding the buffer. Naturally, this field is not present in non-banked systems.

LINK contains the address of the next BCB in the linked list. If this is the last BCB in the list, this field will contain zero. This field isn't present on non-banked systems as only one buffer and associated BCB exist.

Having covered the DPH, XDPH and associated data structures, we can now examine the Disk Parameter Block in detail.

Disk Parameter Blocks

The CP/M 2.2 DPB

The disk parameter blocks in the BIOS tell the BDOS all it needs to know about the physical characteristics of the disk. The CP/M 2.2 DPB for a Gemini QDDS format disk looks like this:

```
Hex    Dec    Interpretation
0028H   40  ; SPT - CP/M Sectors (128 bytes) per track
  05H    5  ; BSH - Block Shift Factor
  1FH   31  ; BLM - Block Mask
  03H    3  ; EXM - Extent Mask
00C4H  196  ; DSM - Disk Size in blocks-1
007FH  127  ; DRM - Number of directory entries-1
  80H       ; ALO - Reserved directory..
  00H       ; AL1 - ..blocks
0020H   32  ; CKS - Checksum Size
0002H    2  ; OFF - Number of system tracks on disk
```

The abbreviations used are those used by Digital Research in the CP/M 2 Alteration Guide. Decimal values are given where appropriate. Note that CP/M 1.4 doesn't use Disk Parameter Blocks and that the DPB for CP/M 3 contains extra information about physical sector size.

CP/M uses the value SPT to calculate the track and sector numbers for a particular disk data block. The block number and an offset showing where in the block the required record lies need to be converted so that the appropriate track and sector can be accessed. SPT is determined by the physical disk format. Gemini in the QDDS format have 10 sectors of 512 bytes on each track. CP/M uses 128 byte records and so 40 CP/M sectors will fit on a track. The Gemini SDDS format disks use a figure of 18 for SPT while the Gemini DDDS format uses 80. The latter figure is possible because of the method of access to the disk for the DDDS format. Some explanation seems in order.

Cylindrical and Track methods of disk access

There are two methods of accessing disks. The first method is to start on side 1 of the disk and number the tracks from say 0 to 79 (QDDS) then flip sides and call the tracks 80 to 159. Note that this is just for convenience as the tracks on side 2 of the disk are also physically numbered 0-79. As far as CP/M is concerned, it is seeing a single sided disk with 160 tracks. This is known as the Track method of accessing a disk.

The second method is to have the tracks on both sides of the disk numbered from say 0 to 34 (DDDS) and having filled track 0 on side 1 we flip sides and continue writing to track 0 on side 2 of the disk. In this case CP/M thinks it seeing a single sided disk with 35 tracks but the tracks are twice as long as before (i.e. 80 SPT). This method of access is known as the Cylinder method. You should note that the Gemini Quad density format is a misnomer used by a number of companies. In reality, it should be called 80 track double density or double density with double track density.

BLM, BSH and EXM

The block mask (BLM) is basically one less than the number of 128 byte records that will fit in one block. Our figure of 31 means that 32 records will fit in the block. This enables the BDOS to calculate the block size as, you will have noticed, it isn't given in the DPB. If we take the current record count from the FCB and logically AND it with the block mask, we will obtain a number in the range 0-31 which tells us where in a block the particular record may be found. Take, for example, that the current record count is 92 or 01011100. If we AND that figure with 31 or 00011111 we get the result 00011100 or 28. This indicates that the record occupies the 29th (0-31 remember) record position in the block.

The extent mask (EXM) is one less than the number of extents that may be controlled by one directory entry. This depends upon the maximum disk size as, if 16 bit values are recorded in the directory, then each directory entry will refer to half the number of extents than it would if 8 bit entries were being used. In the case of the QDDS format, this value is 3 which indicates that 4 extents may be recorded in a single directory entry. When used as a mask by ANDing the extent byte of the directory entry with EXM, we produce a number in the range 0-3 which tells the BDOS which extent we are referring to in the entry.

The next field is the Block Shift Factor (BSH), and is used by the BDOS when converting CP/M records and extents into block positions in a directory entry and vice versa. BSH is defined as the logarithm base two of the block size in 128 byte records or, LOG2(BLS/128) where LOG2 represents the binary logarithm function. That last bit is more or less a direct quote from Digital Research and probably means as much to you now as it did to me initially.

A beginners guide to base 2 logarithms

For those of you who, like me, left school with Maths 'O' level or alternatively finished school a long time ago (or both), here is a quick refresher. The notation that I shall use to represent powers is borrowed from Microsoft MBASIC. e.g. ten squared will be shown as 10^2 while ten cubed will be shown as 10^3.

Consider the following:

Decimal	Binary	Equivalent
1 = 10^0	1 = 2^0	= 1
10 = 10^1	10 = 2^1	= 2
100 = 10^2	100 = 2^2	= 4
1000 = 10^3	1000 = 2^3	= 8
10000 = 10^4	10000 = 2^4	= 16
100000 = 10^5	100000 = 2^5	= 32
1000000 = 10^6	1000000 = 2^6	= 64

These are simply powers of 10 and powers of 2. The similarities between the two number systems are quite evident and most, if not all, assembly language programmers should have seen individual bits in a byte referred to as powers of 2. The useage of powers of two in all probability explains why the bits in a byte are numbered from 0 to 7 rather than 1 to 8.

The relationship between logarithms and powers is extremely close as can be seen from the examples below:

log10 of 1 = 0	log2 of 1 = 0
log10 of 10 = 1	log2 of 2 = 1
log10 of 100 = 2	log2 of 4 = 2
log10 of 1000 = 3	log2 of 8 = 3
log10 of 10000 = 4	log2 of 16 = 4
log10 of 100000 = 5	log2 of 32 = 5
log10 of 1000000 = 6	log2 of 64 = 6

Logarithms of intermediate values do not fall on a linear scale. i.e. log10 of 50 is not 1.5 but is, in fact, 1.69897 to five decimal places. That is to say that 10^1.69897 = 50. Base 2 logarithms follow exactly the same rules as for base 10 logs. Log2 of 3 is 1.58496 to five decimal places. i.e. 2^1.58496= 3.

The integral part of a logarithm is known as the characteristic and the fractional part is known as the mantissa. For the purposes of explaining BSH, only the characteristic of the base 2 logarithm is required. To simplify matters further, Digital Research have kindly done the calculations for us so if you came out of the last couple of paragraphs with a headache, don't worry.

How BSH, BLM and EXM are used

For completeness, I have copied the tables for the values of BLM and BSH provided by Digital Research. The values, as may be expected, are dependant upon block size (BLS).

BLS	BSH	BLM
1024	3	7
2048	4	15
4096	5	31
8192	6	63
16384	7	127

Having obtained values for BSH from the table (or by calculation), what are they used for? Below is a simple program in BASIC that will calculate X/256 with a remainder. This program is normally used to calculate values for a 16 bit POKE (i.e. to two consecutive memory locations) where X may contain any value between 0 and 65535.

```
10 INPUT "Address to be POKED";ADDRESS
20 INPUT "Enter value to POKE";X
30 P1 = X AND 255
40 P2 = INT(X/256)
50 POKE ADDRESS,P1
60 POKE ADDRESS+1,P2
70 END
```

If we analyse this program and take the binary equivalents of the numbers, this program explains, in precise terms, the relationship between BLM and BSH.

Imagine that the number entered for X is 273. Follow the sequence below:

```
0000000100000111        ; 273 in binary (16 bit)
0000000011111111        ; 255 in binary
0000000000000111        ; 7 (Result of logically ANDing the two
                        ;    values together) = P1

0000000100000111        ; 273 again
       ^                ; This bit has the value 256 (2^8)
0000000000000001        ; 1 (Result of integer dividing by
               ^        ;    256) = P2
```

In the division, the 256 bit (2^8) has moved to the 1 position (2^0). In effect we have right shifted the number 8 places. Another way of looking at it is that we have right shifted the number log2 of 256 places.

In the example DPB given, BLM has a value of 31 and BSH has a value of 5. Their use is exactly the same as in the BASIC program.

Imagine that we wish to read a 128 byte record from disk and that record is number 43 in extent 18 of the file we are reading. Assume that the correct directory entry is available and that it looks something like the one given below:

```
00544553 54202020 20545854 13000080   *.TEST    COM....*
202E2F30 31323334 35363738 393A3B3C   *-./0123456789:;<*
```

We shall use the values for the Gemini QDDS format in this example and so we can see that the directory entry controls four extents. For convenience we shall number the extents 0 to 3. Extent 0 has blocks 2D, 2E, 2F and 30 allocated to it; extent 1 has blocks 31, 32, 33 and 34 and so on for the remaining two extents. Within each extent, we shall also number the block positions from 0 to 3 for convenience.

Since we are using a 4K block size, each block will contain 32 CP/M records of 128 bytes each. We shall number those records 0 to 31.

If we logically AND the required extent (18) with the extent mask (3) we obtain the value 2. This means that the record we require is contained in the extent numbered 2 in our directory entry. i.e. the record is held in one of blocks 35, 36, 37 or 38.

If we now take the record number (43) and right shift it BSH (5) times, we are left with the value 1. The block in position 1 is in fact block number 36 on the disk.

All we need to do now is to determine where in block 36 the particular record we require is located. This is achieved by ANDing the record number with the block mask. The result of ANDing the record number (43) with BLM (31) is 11. Therefore, the record we require is the 12th record in the block.

Back to the DPB

The value DSM contains the number of blocks on the disk less one. Another way of looking at it is that it holds the number of the highest block since the blocks are numbered from 0 to DSM or, in our case 0 to 196 (0-0C4H).

DRM contains the number of directory entries on the disk less one. Again, the directory entries are numbered from 0 to DRM (0-127 or 0-7FH for the QDDS format).

AL0 and AL1 contain the first two bytes of the allocation vector for the drive for which the DPB applies. These two bytes are copied into the allocation vector when the drive is logged in. Remembering that the directory occupies the first block or two on the disk, we need to reserve the required number of blocks by filling in the appropriate bits in the allocation vector. In our case, we only need to reserve one block and this has been done by setting bit 7 of AL0, hence the value 80H. Clearly, we can now see that a maximum of 16 blocks may be reserved for directory entries (by setting all 16 bits in AL0 and AL1). With a 1K block size, a maximum of 512 directory entries may be allocated at 32 directory entries in each block. If a 16K block size is used, the maximum number of directory entries will now be 8192.

CKS is known as the check size and simply tells the BDOS the length of the checksum vector for the currently logged in drive. If fixed media drives are in use, the value of CKS may be set to zero to prevent directory checking.

OFF is the number of system tracks on the disk. This figure depends upon the physical disk format and the CP/M system size. Clearly, a single density disk will require more system tracks than a double density disk in order to save the system. As an example, the SDDS format has 3 system tracks, the QDDS and QDSS formats have 2 and the DDDS format has 1. The discrepancy between the QD and DD formats is again because of the apparent length of the tracks in the DD format which uses the cylindrical method of disk access. CP/M uses the OFF parameter when calculating track numbers for a pending read or write so that it effectively ignores the system tracks. This is particularly useful when partitioning a high capacity drive (i.e. a Winchester) into a number of logical drives. If, say, the first 50 tracks are defined as logical drive A, logical drive B could have a value of 50 for the OFF parameter so that it would start at the 51st track of the physical drive.

The CP/M 3 DPB

Now that we have exhausted the CP/M 2.2 Disk Parameter Block, we can now examine the CP/M 3 DPB. An example of the latter is given below:

```
Hex    Dec    Interpretation
0028H   40  ; SPT - CP/M Sectors (128 bytes) per track
  05H    5  ; BSH - Block Shift Factor
  1FH   31  ; BLM - Block Mask
  03H    3  ; EXM - Extent Mask
00C4H  196  ; DSM - Disk Size in blocks-1
007FH  127  ; DRM - Number of directory entries-1
  80H       ; AL0 - Reserved directory..
  00H       ; AL1 - ..blocks
0020H   32  ; CKS - Checksum Size
0002H    2  ; OFF - Number of system tracks on disk
  02H    2  ; PSH - Physical Record Shift Factor
  03H    3  ; PHM - Physical Record Mask
```

A brief examination of the CP/M 3 Disk Parameter Block will reveal that it is identical to the CP/M 2.2 DPB apart from the last two entries so I shall only explain these additions.

PSH is the physical record shift factor and is described by Digital Research as being LOG2(physical sector size/128). For the Gemini QDDS format which uses 512 byte physical sectors, this value is 2.

PHM is the physical record mask and is calculated to be (physical sector size/128)-1.

PSH and PHM are used in exactly the same way as BLM and BSH except that instead of converting CP/M records and extents to block numbers, they are now converting CP/M records to physical sector numbers. Imagine that we have extracted a block and CP/M record number from the directory using BSH, BLM and EXM. The value that we obtained when explaining BSH and BLM were block number 36 on the disk and record number 11 within that block.

There are 32 CP/M records in the block numbered from 0 to 31 as described before. Additionally, we now know that there are eight physical sectors (of 512 bytes each) in the block, each containing

four CP/M records. We can number these physical sectors from 0 to 7 and the CP/M records within them from 0 to 3.

If we shift the record number (11) right PSH (2) times we get the value 2. It is a simple matter to read the third physical sector into the deblocking buffer ready to extract the required 128 byte record.

Finally, by ANDing the record number (11) with PHM (3) we get the value 3. Consequently, the required record occupies the last 128 bytes in the physical sector, i.e. the fourth record in the sector.

Blocking and Deblocking

Isolating separate 128 byte CP/M records in a physical sector is central to the concept involved with blocking and deblocking. All versions of CP/M transfer data to and from the disk in 128 byte chunks (including CP/M 3). We have already seen that disks may be formatted into sectors of varying size and if a sector size of 128 bytes is used then no problems are experienced. However, many disk formats use sector sizes of 256, 512 and 1024 bytes. As we have seen, Gemini use a 512 byte sector size for their DDDS, QDSS and QDDS disk formats. Since CP/M requires 128 byte records, surely it is easier to format the disk in 128 byte sectors.

The answer to this is yes. It is easier to use 128 byte sectors but oddly enough it is generally faster to use larger sector sizes. Why this is will become apparent as we proceed but suffice to say that since we are using sectors that are larger than 128 bytes, we need a method of providing CP/M with 128 bytes at a time and allowing CP/M to write 128 bytes at a time. The processes for writing and reading in this manner are known as blocking and deblocking respectively.

Blocking and deblocking are implemented in the BIOS under CP/M 2.2 and (usually) in the BDOS under CP/M 3 although facilities exist within CP/M 3 to allow the BIOS to do this. The method used requires a buffer in RAM that is one physical sector long (512 bytes for the Gemini DD and QD formats). This buffer is simply an area of RAM reserved by the BIOS for this purpose and is known as the blocking/deblocking buffer. This buffer is located within the BIOS under CP/M 2.2 but under CP/M 3, the physical record buffers controlled by the BDOS (pointed to by the DTABCBs) are used. From now on I shall assume that CP/M 2.2 is in use and will describe the method of blocking/deblocking in respect of the CP/M 2.2 BIOS. Under CP/M 3, the BDOS will have calculated physical sector numbers (using PSH and PHM) and the BIOS reads them directly.

When deblocking, the BIOS takes the sector number provided by the BDOS and converts it into a physical sector number. i.e. the physical sector that contains the required CP/M record. This physical sector is then read into the blocking/deblocking buffer and the wanted 128 bytes are identified and passed to CP/M by copying them to the DMA address. At the time of reading the physical sector, certain facts about it are recorded. These are the drive, track and sector number for the physical sector. At the same time a flag is set to say that the buffer is in use.

The next time that a read is requested by CP/M, the physical sector number is calculated as before. This time, however, the BIOS will find that the buffer is in use. Bearing in mind that most disk reads are done sequentially, it is a fair bet that the CP/M record we want is in the buffer already. The BIOS checks the drive, track and sector numbers for the required sector against those stored for the buffered sector and if they match, the BIOS skips the disk read and simply transfers the required 128 bytes to the DMA address.

When reading sequentially using deblocking, the disk is accessed only every fourth request from CP/M. As disk drives are relatively slow when compared with the time required to extract data from RAM, you can now see why this process is faster than simple disk reads using 128 byte sectors. Generally, the larger the physical sector size, the greater the improvement in disk access time will be.

Writing to disk using blocking offers a similar (but not as great) increase in speed. You will see why writing is not as fast as reading in a moment.

Imagine that we wish to write a random record to disk and that we are updating one record among many adjacent records. Since we have to write to the disk in terms of complete physical sectors, we cannot simply write 128 bytes to the disk. If we put the CP/M record in the blocking/deblocking buffer and wrote that to disk, we would erase the other three records in the physical sector.

When writing CP/M records to disk, the BIOS computes the physical sector number in the same manner as for reading. It then checks to see if that physical sector is already in the buffer and reads it if not. This pre-read of physical sectors explains why the disk performance is not quite as good when writing as compared with reading. However, the performance achieved is still better than when writing 128 byte sectors as the disk is still only accessed twice for every four CP/M records written.

The pre-read may be skipped under certain circumstances. CP/M tells the BIOS that the data being written lies in an as yet unallocated block. If this is so, there is no data in the sector to be destroyed so the read is skipped. If the pre-read is required, before reading the sector into the buffer, there is another check that must be done by the BIOS as we shall see in a moment.

Having read the required physical sector, the buffer in use flag is set and the CP/M record may be put in it at the appropriate place. You will see that the other records in the sector have now been preserved. The BIOS doesn't write the buffer to disk at this point as another write to the same physical sector may be requested by CP/M, so it simply flags this record as containing unwritten data and returns control back to CP/M.

If another write is requested to the same sector, the pre-read of the disk will not be required as the wanted physical sector will already be in the buffer. However, if another write (or read) is requested that involves a different physical sector, the BIOS looks at the flag showing whether the buffer contains unwritten data and if it does, the BIOS now writes it to disk.

The only exception to the above explanation is when directory information is being written to disk. Since the BIOS maintains a separate buffer for physical directory sectors, the changed directory information MUST be written to disk immediately.

THE END !

A Review of two Modula 2s by Doug Taylor

If you liken programming languages to cars, then Cobol is an Austin 7, old fashioned and requiring a lot of maintenance, BASIC is a Fiat 127, noisy, prone to rust and always running out of performance when you need it. Pascal is the Volvo of the programming world, careful, safe and always willing to cruise up the hard shoulder at 90 MPH. Whereas Fortran 77 is a Range Rover, robust go anywhere, do anything, break or remake all the rules language.

So where does Modula 2 fit? Well this is a kit car, built with strength of a Volvo - it is after all a member of the Algol family, a child of Nicklaus Wirth, but has all the versatility of Fortran. Modula 2 has been used by Wirth to correct many of the mistakes of Pascal, the rather fussy and confusing syntax has been tidied up, (I could never remember when I needed a semicolon and when I didn't) and the most powerful feature of Fortran, the foundation in library based code, imported.

Modula 2, as it name suggests, encourages you to write modular code. These modules are combined into libraries, and programs created by importing code into the program from these libraries. The library will usually consist of Modula 2 code, it is a very recursive language, Modula 2 code is made from Modula 2 code which is made from Modula 2 code which is ...., but in most if not all applications, code from other relocatable code producing tools can be linked into the program, e.g. from an assembler or Fortran Compiler.

The two versions of Modula 2 I will review are Turbo Modula 2 from Borland International (only available for CP/M at the moment) and FTL Modula 2 from Workman and Associates (HiSoft in the U.K.) which is available for MSDOS and CP/M, the version reviewed being the MSDOS variant.

The Borland version of the language comes complete with a 544 page manual describing the language, the Wordstar like editor, the shell and the Standard library supplied. The standard library and reference directory to the modules is explained briefly with examples, covering 411 pages of the manual. If you have used Borland's other minor work Turbo Pascal, you will know the high standard of their manuals - all examples in the manual will work and you can learn the language from these alone.

Making CP/M More User Friendly by C. Bowden

This article briefly discusses the software available to improve CP/M and suggests some simple CBIOS modifications that the user can carry out, that will give additional system flexibility. These modifications apply to CP/M Version 2.

Only a day or two ago an acquaintance who is still using NASDOS and POLYDOS rang me and said 'You use CP/M don't you - I have heard that its as friendly as [expletive deleted] so I don't know whether to upgrade'. My reply was to the effect that since he was a Nasbus/80-BUS user, he could progress to what is probably one of the best implementations of CP/M available.

Certainly, to the average user of microcomputers, the mouse and window approach as exemplified in the Mackintosh is very attractive and easy to use. However I think that most readers of this newsletter are probably more aware of the inner workings of the machine than the average user, and would find the relative inaccessibility of the modern machine extremely frustrating.

The NASBUS/80BUS system is geared to the engineer, scientist or enthusiast who needs to be able to alter his hardware systems and associated software. The system may look a little old fashioned when compared with the sleek plastic machines around now, but it lives on whilst many others fall by the wayside. The enormous range of available CP/M software (and our investment in it), and the flexibility of systems like ours make it worthwhile to stay in the 8 bit world.

Of course standard CP/M is rather unfriendly. It evolved in a world where TTY terminals, Tape readers and Punches were still common. This is still reflected in software like ED.COM and MBASIC.COM, where the line editing features are truly as unfriendly as a hungry wolf. Fortunately, it is not necessary to remain locked in the embraces of standard CP/M. There is a lot of software around, much of it in the public domain, that can transform CP/M into a much more sophisticated system, and the various CBIOS's available on Nasbus/80-BUS systems make system extremely friendly when compared to many other systems. Above all, it is the 'ON SCREEN' EDIT feature, starting back in the good old NASCOM days, that is so useful.

CP/M consists of three modules, namely the CCP, the BDOS and the BIOS. These modules are much more fully described in the CP/M manuals, various books and some of the references in Appendix 2.

The CCP (Console Command Processor) is the part of CP/M that sits showing the A> prompt and waits for your command. It is 2k bytes long.

The BDOS (Basic Disk Operating System) is the interface between the CCP or currently running program and the BIOS. It is 3.5k bytes long.

The BIOS or CBIOS (Customized Basic Input Output System) holds the software that actually controls the system hardware. The CBIOS therefore varies from machine to machine. It can be any length up to about 4k Bytes long. Obviously the longer the CBIOS, the better the system (ought to be) in terms of hardware support and user friendliness.

It is not intended to describe in any detail the implementation or modification of advanced modules, which is usually well described in any accompanying documentation, but merely to describe what is available, and to indicate the main features. Readers who wish to obtain a more detailed understanding of CP/M or modifying it may find some assistance in articles referred to in Appendix 2

Improving the CCP - CCPZ/ZCPR2

A number of very useful improvements can be made to the CCP. The standard CCP provides six built in commands (DIR, ERA, REN, SAVE, USER and TYPE). There is no screen paging support, no 'PATH', and precious little else.

There are two alternative software replacements for the CCP. The simplest approach to system improvement is to replace the standard CCP with CCPZ or earlier versions of ZCPR. These programs are virtually the same and are based on ZCPR in the SIG/M or CP/M User Group Library. ZCPR literally means Z(80) CCP Replacement. Since Z80 code is usually more compact than 8080 code, it is possible to include more in the available 2k. The resulting new CCP is compatible with the standard CCP, and provides all of the standard facilities. It also provides the following extra features which are more fully described in the .DOC files provided from the library.

a) Displays the names of ALL files removed by an ERA command.
This can save potentially fatal errors.

b) Provides improved Directory display and SYS/DIR File options.
A)ll and S)ys options for DIR+SYS or SYS file display. Better directory formats may be selected for assembly.

c) Optionally displays the current USER number in CP/M prompt.
This is an assembly option.

d) Optionally provides screen paging support for screen output.
May be assembled to default to on or off. `P' parameter will toggle this option. (Normally off on this system as BIOS will provide better paging.)

e) Allows default USER area to be altered, by DFU command. eg: DFU 3
This would mean that the `path' would search USER 3 and not 0. The USER number may be given in Decimal or Hex. eg: DFU FH.

f) Provides a GET command to load a file to memory.
Eg: GET 8000 MYFILE.COM would load MYFILE.COM in memory at 8000H.

g) Provides JUMP and GO commands to operate a TPA resident program.
JUMP will `call' a subroutine. eg: JUMP E000H.
GO will call the subroutine at 100H. eg: Restart MBASIC etc. GO is the same as JUMP 100H.

h) The SAVE command allows HEX or DECIMAL, PAGES or SECTORS.
eg: SAVE 18 TEST.COM or SAVE 12H TEST.COM to save 18 pages. You may also give Number of sectors with `S' parameter. eg: - SAVE 10H ANOTHER.ONE S or SAVE 16 ANOTHER.ONE S for 8 Pages. N.B. 256 byte `page' or 128 byte `sector'.

i) More flexible handling of SUBMITS and CCP buffer default Cold Boot Commands.

j) Provides a LIST command to send a file to the printer.
eg: LIST THISFILE.TXT. The CBIOS will page the printer.

k) Provides a search path for required .COM files.
This is so useful that it merits a fuller description.

With a normal CCP if you issue a command like `STAT' then CP/M will look for STAT.COM on the current drive and USER area. If the file is not found, CP/M will give up with a query `STAT?'.

With CCPZ/ZCPR a three level search is performed. The CCP will look in the current user area of the default drive. If the file is not found, user area 0 of the default disk is searched. If this fails, USER 0 of drive A: is searched. If the file is still not found, CP/M will give up with the usual query. This

feature is extremely useful even on a two floppy system, but is invaluable in systems with Virtual or Winchester discs. (If the default USER is changed by the DFU command, the new default USER will be searched instead of USER 0 in the above example.)

NOTE that CCPZ has been standard with all Gemini CP/Ms for some time now.

Improving the CCP - ZCPR3

A second method of CCP improvement is possible by the use of the more recent versions of ZCPR. In addition to replacing the CCP, additional memory is used above CP/M and a considerable system enhancement is obtained. App. 2 Ref. 10 describes the system more fully. I am hoping to try ZCPR3 in the near future, but I have not used it yet so I can only indicate a few of the features :-

Environment Descriptor - Holds information on the ZCPR3 system.
Named Directories - Allows Drives and USER areas to be named.
Commands -
Resident Command CCP - GO, SAVE, GET, JUMP.
R.C.P. Segment - CP, ERA, TYPE, LIST, PEEK, POKE, PROT, REN.

Flow Control - Allows conditional testing of CCP processing.
Input/Output Package - Routes I/O and redirects data.
Terminal Descriptor. - Describes the Terminal and its commands.
Five level search PATH. (Easily redefinable)
Wheel Byte - Improves system security.
Drive and USER access via a C6: type command.
Support is provided for security via Passwords.
Large number of support utilities like HELP, MENU, SHOW, UNERASE, CONFIG, CLEANDIR etc;

One penalty of ZCPR3 is that some additional system memory is needed which reduces the available TPA by about 4k. If this were a problem with certain programs, a more standard operating system could be loaded with such programs.

Improving the BDOS - BDOSZ

This area of CP/M is the least amenable to modification. However a BDOS replacement known as BDOSZ is available, and improvements have been incorporated by using Z80 code to provide extra room for refinements in the 3.5k of space available. BDOSZ provides the user with much better recovery from errors than the standard BDOS. A summary of features that are provided by BDOSZ :-

a) When a 'SELECT ERROR' occurs you may enter a valid drive No. and continue.

b) When a Disk is 'R/O' you will be asked 'DO IT ANYWAY'. If you answer 'Y' BDOSZ will reset the disk and proceed.

c) If a 'R/O' file is found during ERA, the screen will display the file name and the same query will be printed. If you answer 'Y' the file R/O flag will be reset, and the ERA carried out. This applies to REN and SAVE activities as well.

d) If a bad sector is found, a 'READ' or 'WRITE' error message will be displayed. ^C will cause a Warm Boot, but any other key will cause the error to be ignored, so that it is possible to recover partly bad files.

e) DISK or DIRECTORY FULL errors. BDOSZ will display a 'Change Disk Y/N/^C' option and allows the Disk to be changed if desired.

BDOSZ is a worthwhile improvement to CP/M. (N.B. I have found a Bug - see note in App 1.)

Customizing the CBIOS

The CBIOS is the area where most can be done to make CP/M more user friendly. I recently bought an Alphatronic PC - A Z80 CP/M machine that was very well made and selling at a bargain price. I had thought that it would be useful as a second machine. I kept it for about six months, and then sold it to a friend who had previously been using a PET. He thinks that it is fabulous, so why did I sell it ? (I did put CCPZ onto it.)

Well, I found that I just could not put up with a machine that had no Type Ahead, no screen dump, no screen editing, limited paging (from CCPZ), limited function key support, no backspace key, kept changing 'case', and had partly ROM based BIOS. The supporting utilities for copying and formatting were not very good either - pretty screen displays, but no verification, no information on progress, and a bug or two. At least when I sold it it had CCPZ and some public domain utilities to help. It made me realize just what some people have to put up with.

A number of BIOS's are available for the Nasbus/80-BUS machines. I have used BIOS's by Gemini, Richard Beal (SYS) and MAP80 Systems. I have no knowledge of Nascom CP/M BIOS's so I cannot comment on them. The three BIOS's mentioned are all very good and provide features that are rare, if not unique, on 8 Bit CP/M machines. Fortunately, MAP and SYS BIOS's come/came with source code. (Unfortunately SYS cannot now be purchased, but I believe that there are quite a few about.)

If I can ride one of my hobby-horses here - Whilst I fully understand the problems of copyright and pirating, I feel that it is a completely retrograde step to withhold the BIOS source code from system purchasers. The vast majority of users are being penalized for the sake of the odd dishonest person who could probably be dealt with effectively by the law anyway. I feel that piracy is being made the excuse for trying to prevent people from expanding their systems except the Gemini way. SYS at one stage included Winchester support, but later this had to be removed. (Because users didn't need to go to Gemini for a new BIOS or to buy a Winchester ?) Then SYS itself was withdrawn because of copyright problems. (See App. 2, Ref 4)

The new GM849 card seems to me to be a similar case in some ways. I recently wanted to purchase a spare Disk Controller card for several machines at work that are in heavy use, and I was annoyed to find that the GM849 card that supercedes the GM829 needs version 3.4 of the BIOS to run it. (When the CP/M was upgraded to Version 3.2 BIOS at work a few months ago there was no mention that it would not drive the 849 - nor was there a mention of the 849). The CP/M on the machines in question cannot be easily replaced due to special custom routines, so I will have to try to alter the CBIOS to support the 849. This sort of work I can do without.

Logically one would expect that Gemini would want to offer the best BIOS available, but their BIOS does not include a number of useful features such as locked EDIT and extended Screen Paging, and without the source code, it is of course impossible to customize it for additional features such as Clock support, or the type of keyboard feature described later. Neither are MAP products supported, again denying users of the BUS the most flexible system. Once one is accustomed to such features, it would be difficult to return to a BIOS without them. I appreciate that there must be limits to what can be provided as 'CONFIG' options, but I do not see that as any reason to deny the users ie: CUSTOMERS, the facility to do their own customizing.

Fortunately I have been able to combine many of the features of SYS with the MAP CP/M 2.2 BIOS, and then added in some of my own routines to provide myself with the features that I want. (For personal use only.) Customization like this takes a lot of time though, and obviously would not be possible for many users, even if the required source codes are available. The restrictive approach must be denying users the best operating environment, and thus could be detrimental in the long run.

Enough of the complaining - If Gemini BIOS 3.x is what you have it is still a lot better than most CP/M BIOS's. If you are fortunate enough to have SYS18 then you will have everything that you need except possibly Winchester support. The MAP CP/M BIOS is very good and easily adaptable to different disk types and formats, including Winchesters, but lacks a few features like screen dump, screen paging and VBOOT, which can be added anyway.

If you do have the source code of your BIOS available then you may like to modify it to include some simple but extremely useful keyboard activated hardware support. The rest of this article will describe the sort of thing that I have added to my BIOS.

I first had the idea of providing keyboard support to hardware in order to cause the printer to advance to top of next page.

In order to permit a direct keyboard command it is necessary to trap and process the relevant keystrokes instead of passing them to CP/M. This means that it is necessary to modify the lowest level BIOS routine reading the keyboard. The 'special' keys must be intercepted and processed and then control returned smoothly to CP/M.

The first problem was to find a suitable key for the purpose. After a long perusal of key allocations I decided to use ^T. This choice was based on the fact that few programs used ^T and it could be dispensed with, and also that `T' was a synonymous with printer T)hrow. The keyboard routine (pkbd: in SYS and BLINK: in MAP) was accordingly modified so that if ^T was typed, the printer advanced to top of next page, and the BIOS lines per page counter was reset.

It did not take long to realize that I could utilize this method to provide a number of other simple features that could avoid me loosing what little hair I have left. Due to the shortage of key allocations, I decided to make ^T the lead-in key, and to program the software to expect a second control key depending on the desired function. This had the advantage of allowing a key to be chosen, that related to the function to be achieved. In addition, it allowed the option of sending on a ^T to CP/M if desired.

As an aid to the user, a message is displayed on the locked top line of the screen whilst the system is waiting for the second character, reminding the user of the options available. After the second character has been typed, this message is erased. After the routine has been processed, a CR character is returned to CP/M. This also happens if an erroneous second character is typed. Keyboard software is not noticeably affected since only the first ^T is searched for. Any other character is returned normally to CP/M.

A problem became apparent a while after I added a Real Time Clock to my BIOS. (I have described this in some detail - see App. 2. Ref 5). Since the article was published I discovered that if the clock updated during screen EDIT, or operation of screen oriented software, (eg: Cursor Addressing), the Clock could interfere with the 'ESC' sequences and corrupt the screen. At first I solved this problem by disabling the clock or running a BIOS without the clock. Since this resulted in incompatibilities in the various CP/M systems, I later decided to make the clock `passive' and provide three ways of updating it.

The display now updates on Warm/Cold Boot. On CALL from an external program. By user initiated demand from the keyboard as described below.

This has eliminated the screen corruption problems referred to above. The number of functions supported is optional. It is currently limited in my case by the BIOS size equalling the 4k of BIOS space available on the system track. I have at present added seven functions as direct keyboard commands. They are :-

a) ^T,^D - Call clock routines and Update Date and Time display on Locked top line at Left hand side of the screen.

b) ^T,^P - Send a Form Feed to printer, causing it to advance to top of next page, and reset BIOS lines per page count.

c) ^T,^R - Reset IVC/SVC to 80 wide. Useful for the (very) rare time when screen corrupts or gets out of vertical sync.

d) ^T,^N - Switch attached (Epson) Printer to Normal Print.

e) ^T,^C - Switch Printer to Compressed Mode.

f) ^T,^S - Reset Screen Paging flag to 0, to enable paging if a 'W' or 'K' was issued and a Warm/Cold Boot is undesirable.

g) ^T,^T - Pass ^T on to calling program.

The listing above, on pages 49 and 50, shows in upper case part of the original BIOS code (MAP BIOS), and lower case shows the modifications, but note that the cursor definitions have been moved into CURON/CUROFF to fit in with other BIOS modifications.

If there is plenty of spare space in the BIOS, quite a number of printer support features could be incorporated, and even function key redefinitions.

Appendix 1

I have detected a problem with BDOSZ. Recently I have added a Winchester disk to a Gemini system. Despite the fact that CP/M Plus is available, CP/M 2.2 on the machine in question has been customized extensively along the lines of this article and this operating system is still preferred for many purposes. Consequently I needed compatibility between CP/M 2.2 and CP/M Plus in Winchester support. I eventually cobbled together a CBIOS with all of the required features and compatible Disk format by using a couple of MAP CP/M 2.2 BIOS's and some routines from SYS.

CP/M 2.2 at that stage had CCPZ and BDOSZ in operation. I then found that both CP/M 2.2 and CP/M Plus were treating the floppy drives the same in respect of the way the Directory Block allocation was being written, but wrote them differently to the Winchester. CP/M Plus was putting a 00 byte between block numbers on the Winnie directory, but CP/M 2.2 was not. After reverting to standard CP/M 2.2 BDOS the Directory on the Winnie was the same on both versions of CP/M. The 00 byte is not present on the floppy directory under either operating system but present on the Winnie under both. If anyone knows of a patch for BDOSZ to make it treat the Winnie properly, I would be pleased to obtain details.

Appendix 2

Some useful References:

1)	Disks and CP/M	I.N.M.C.80 News No. 5
2)	Customizing your CBIOS	80BUS News. Vol 2, Iss 1
3)	SYS-Latest Developments	80BUS News. Vol 2, Iss 1
4)	SYS is dead - long live ?	80BUS News. Vol 2, Iss 4
5)	CBIOS Real Time Clock	80BUS News. Vol 3, Iss 6
6)	CP/M Features & Facilities	CPMUG U.K. Vol 1, No 1.
7)	The BIOS	CPMUG U.K. Vol 1, No 5.
8)	The Console Command Processor	CPMUG U.K. Vol 1, No 8.
9)	ZCPR Replacement CCP	CPMUG U.K. Vol 2, No 2.
10)	The SB180 - Software	Byte. Oct 1985
11)	Soul of CP/M	Sams & Co. (U.S.A.)
12)	Mastering CP/M	Sybex. (U.S.A.)
13)	CP/M manuals.	Digital Research.