How It Works -- CHS Translation

Plus BIOS Types, LBA and Other Good Stuff

Version 4a

by Hale Landis (landis@sugs.tware.com)

THE "HOW IT WORKS" SERIES

This is one of several How It Works documents. The series currently includes the following:

* How It Works -- CHS Translation
* How It Works -- Master Boot Record
* How It Works -- DOS Floppy Boot Sector
* How It Works -- OS2 Boot Sector
* How It Works -- Partition Tables

Introduction (READ THIS!)

This is very technical. Please read carefully. There is lots of information here that can sound confusing the first time you read it.

Why is an understanding of how a BIOS works so important? The basic reason is that the information returned by INT 13H AH=08H is used by FDISK, it is used in the partition table entries within a partition record (like the Master Boot Record) that are created by FDISK, and it is used by the small boot program that FDISK places into the Master Boot Record. The information returned by INT 13H AH=08H is in cylinder/head/sector (CHS) format -- it is not in LBA format. The boot processing done by your computer's BIOS (INT 19H and INT 13H) is all CHS based.

Read this so that you are not confused by all the false information going around that says "LBA solves the >528MB problem".

Read this so that you understand the possible data integrity problem that a WD EIDE type BIOS creates. Any BIOS that has a "LBA mode" in the BIOS setup could be a WD EIDE BIOS. Be very careful and NEVER chage the "LBA mode" setting after you have partitioned and installed your software.

History

Changes between this version and the preceeding version are marked by "!" at left margin of the first line of a changed or new paragraph.

Version 4 -- BIOS Types 8 and 10 updated.

Version 3 -- New BIOS types found and added to this list. More detailed information is listed for each BIOS type. A section describing CHS translation was added.

Version 2 -- A rewrite of version 1 adding BIOS types not included in version 1.

Version 1 -- First attempt to classify the BIOS types and describe what each does or does not do.

Definitions
----

* 528MB - The maximun drive capacity that is supported by 1024 cylinders, 16 heads and 63 sectors (1024x16x63x512). This is the limit for CHS addressing in the original IBM PC/XT and IBM PC/AT INT 13H BIOS.

* 8GB - The maximum drive capacity that can be supported by 1024 cylinders, 256 heads and 63 sectors (1024x256x63x512). This is the limit for the BIOS INT 13H AH=0xH calls.

* ATA - AT Attachment -- The real name of what is widely known as IDE.

* CE Cylinder - Customer Engineering cylinder. This is the last cylinder in P-CHS mode. IBM has always reserved this cylinder for use of disk diagnostic programs. Many BIOS do not account for it correctly. It is of questionable value these days and probably should be considered obsolete. However, since there is no industry wide agreement, beware. There is no CE Cylinder reserved in the L-CHS address. Also beware of diagnostic programs that don't realize they are operating in L-CHS mode and think that the last L-CHS cylinder is the CE Cylinder.

* CHS - Cylinder/Head/Sector. This is the traditional way to address sectors on a disk. There are at least two types of CHS addressing: the CHS that is used at the INT 13H interface and the CHS that is used at the ATA device interface. In the MFM/RLL/ESDI and early ATA days the CHS used at the INT 13H interface was the same as the CHS used at the device interface.

   Today we have CHS translating BIOS types that can use one CHS at the INT 13H interface and a different CHS at the device interface. These two types of CHS will be called the logical CHS or L-CHS and the physical CHS or P-CHS in this document. L-CHS is the CHS used at the INT 13H interface and P-CHS is the CHS used at the device interface.

   The L-CHS used at the INT 13 interface allows up to 256 heads, up to 1024 cylinders and up to 63 sectors. This allows support of up to 8GB drives. This scheme started with either ESDI or SCSI adapters many years ago.

   The P-CHS used at the device interface allows up to 16 heads up to 65535 cylinders, and up to 63 sectors. This allows access to 2^28 sectors (136GB) on an ATA device. When a P-CHS is used at the INT 13H interface it is limited to 1024 cylinders, 16 heads and 63 sectors. This is where the old 528MB limit originated.

   ATA devices may also support LBA at the device interface. LBA allows access to approximately 2^28 sectors (137GB) on an ATA device.

   A SCSI host adapter can convert a L-CHS directly to an LBA used in the SCSI read/write commands. On a PC today, SCSI is also limited to 8GB when CHS addressing is used at the INT 13H interface.

* EDPT - Enhanced fixed Disk Parameter Table -- This table returns additional information for BIOS drive numbers 80H and 81H. The EDPT for BIOS drive 80H is pointed to by INT 41H. The EDPT for BIOS drive 81H is pointed to by INT 46H. The EDPT is a fixed disk parameter table with an AxH signature byte. This table format returns two sets of CHS information. One set is the L-CHS and is probably the same as returned by INT 13H AH=08H. The other set is the P-CHS used at the drive interface. This type of table allows drives with >1024 cylinders or drives >528MB to be supported. The translated CHS will have <=1024 cylinders and (probably) >16 heads. The CHS used at the drive interface will have >1024 cylinders and <=16 heads. It is unclear how the IBM defined CE cylinder is accounted for in such a table. Compaq probably gets the credit for the original definition of this type of table.

* FDPT - Fixed Disk Parameter Table - This table returns additional information for BIOS drive numbers 80H and 81H. The FDPT for BIOS drive 80H is pointed to by INT 41H. The FDPT for BIOS drive 81H is pointed to by INT 46H. A FDPT does not have a AxH signature byte. This table format returns a single set of CHS information. The L-CHS information returned by this table is probably the same as the P-CHS and is also probably the same as the L-CHS returned by INT 13H AH=08H. However, not all BIOS properly account for the IBM defined CE cylinder and this can cause a one or two cylinder difference between the number of cylinders returned in the AH=08H data and the FDPT data. IBM gets the credit for the original definition of this type of table.

* LBA - Logical Block Address. Another way of addressing sectors that uses a simple numbering scheme starting with zero as the address of the first sector on a device. The ATA standard requires that cylinder 0, head 0, sector 1 address the same sector as addressed by LBA 0. LBA addressing can be used at the ATA interface if the ATA device supports it. LBA addressing is also used at the INT 13H interface by the AH=4xH read/write calls.

* L-CHS -- Logical CHS. The CHS used at the INT 13H interface by the AH=0xH calls. See CHS above.

* MBR - Master Boot Record (also known as a partition table) - The sector located at cylinder 0 head 0 sector 1 (or LBA 0). This sector is created by an "FDISK" utility program. The MBR may be the only partition table sector or the MBR can be the first of multiple partition table sectors that form a linked list. A partition table entry can describe the starting and ending sector addresses of a partition (also known as a logical volume or a logical drive) in both L-CHS and LBA form. Partition table entries use the L-CHS returned by INT 13H AH=08H. Older FDISK programs may not compute valid LBA values.

* OS - Operating System.

* P-CHS -- Physical CHS. The CHS used at the ATA device interface. This CHS is also used at the INT 13H interface by older BIOS's that do not support >1024 cylinders or >528MB. See CHS above.

Background and Assumptions

First, please note that this is written with the OS implementor in mind and that I am talking about the possible BIOS types as seen by an OS during its hardware configuration search.

It is very important that you not be confused by all the misinformation going around these days. All OS's that want to be co-resident with another OS (and that is all of the PC based OS's that I know of) MUST use INT 13H to determine the capacity of a hard disk. And that capacity information MUST be determined in L-CHS mode. Why is this? Because: 1) FDISK and the partition tables are really L-CHS based, and 2) MS/PC DOS uses INT 13H AH=02H and AH=03H to read and write the disk and these BIOS calls are L-CHS based. The boot processing done by the BIOS is all L-CHS based. During the boot processing, all of the disk read accesses are done in L-CHS mode via INT 13H and this includes loading the first of the OS's kernel code or boot manager's code.

Second, because there can be multiple BIOS types in any one system, each drive may be under the control of a different type of BIOS. For example, drive 80H (the first hard drive) could be controlled by the original system BIOS, drive 81H (the second drive) could be controlled by a option ROM BIOS and drive 82H (the third drive) could be controlled by a software driver. Also, be aware that each drive could be a different type, for example, drive 80H could be an MFM drive, drive 81H could be an ATA drive, drive 82H could be a SCSI drive.

Third, not all OS's understand or use BIOS drive numbers greater than 81H. Even if there is INT 13H support for drives 82H or greater, the OS may not use that support.

Fourth, the BIOS INT 13H configuration calls are:

* AH=08H, Get Drive Parameters -- This call is restricted to drives up to 528MB without CHS translation and to drives up to 8GB with CHS translation. For older BIOS with no support for >1024 cylinders or >528MB, this call returns the same CHS as is used at the ATA interface (the P-CHS). For newer BIOS's that do support >1024 cylinders or >528MB, this call returns a translated CHS (the L-CHS). The CHS returned by this call is used by FDISK to build partition records.

* AH=41H, Get BIOS Extensions Support -- This call is used to determine if the IBM/Microsoft Extensions or if the Phoenix Enhanced INT 13H calls are supported for the BIOS drive number.

* AH=48H, Extended Get Drive Parameters -- This call is used to determine the CHS geometries, LBA information and other data about the BIOS drive number.

* the FDPT or EDPT -- While not actually a call, but instead a data area, the FDPT or EDPT can return additional information about a drive.

* other tables -- The IBM/Microsoft extensions provide a pointer to a drive parameter table via INT 13H AH=48H. The Phoenix enhancement provides a pointer to a drive parameter table extension via INT 13H AH=48H. These tables are NOT the same as the FDPT or EDPT.

Note: The INT 13H AH=4xH calls duplicate the older AH=0xH calls but use a different parameter passing structure. This new structure allows support of drives with up to 2^64 sectors (really BIG drives). While at the INT 13H interface the AH=4xH calls are LBA based, these calls do NOT require that the drive support LBA addressing.

CHS Translation Algorithms

NOTE: Before you send me email about this, read this entire section. Thanks!

As you read this, don't forget that all of the boot processing done by the system BIOS via INT 19H and INT 13H use only the INT 13H AH=0xH calls and that all of this processing is done in CHS mode.

First, lets review all the different ways a BIOS can be called to perform read/write operations and the conversions that a BIOS must support.

! * An old BIOS (like BIOS type 1 below) does no CHS translation and does not use LBA. It only supports the AH=0xH calls:

```
      INT 13H      (L-CHS == P-CHS)             ATA
      AH=0xH  --------------------------------> device
      (L-CHS)                                   (P-CHS)
```

* A newer BIOS may support CHS translation and it may support LBA at the ATA interface:

```
      INT 13H        L-CHS                      ATA
      AH=0xH  --+--> to    --+----------------> device
      (L-CHS)   |    P-CHS   |                  (P-CHS)
                |            |
                |            |    P-CHS
                |            +--> to    --+
                |                 LBA     |
                |                         |
                |    L-CHS                |     ATA
                +--> to  -----------------+---> device
                     LBA                        (LBA)
```

* A really new BIOS may also support the AH=4xH in addtion to the older AH\0xH calls. This BIOS must support all possible combinations of CHS and LBA at both the INT 13H and ATA interfaces:

```
      INT 13H                                   ATA
      AH=4xH  --+-----------------------------> device
      (LBA)     |                               (LBA)
                |
                |    LBA
                +--> to    ---------------+
                     P-CHS                |
                                          |
      INT 13H        L-CHS                |     ATA
      AH=0xH  --+--> to    --+------------+---> device
      (L-CHS)   |    P-CHS   |                  (P-CHS)
                |            |
                |            |    P-CHS
                |            +--> to    --+
                |                 LBA     |
                |                         |
```

```
                |    L-CHS                |     ATA
                +--> to  ----------------+---> device
                     LBA                        (LBA)
```

You would think there is only one L-CHS to P-CHS translation algorithm, only one L-CHS to LBA translation algorithm and only one P-CHS to LBA translation algorithm. But this is not so. Why? Because there is no document that standardizes such an algorithm. You can not rely on all BIOS's and OS's to do these translations the same way.

The following explains what is widely accepted as the "correct" algorithms.

An ATA disk must implement both CHS and LBA addressing and must at any given time support only one P-CHS at the device interface. And, the drive must maintain a strick relationship between the sector addressing in CHS mode and LBA mode. Quoting the ATA-2 document:

```
     LBA = ( (cylinder * heads_per_cylinder + heads )
             * sectors_per_track ) + sector - 1
```

     where heads_per_cylinder and sectors_per_track are the current translation mode values.

This algorithm can also be used by a BIOS or an OS to convert a L-CHS to an LBA as we'll see below.

This algorithm can be reversed such that an LBA can be converted to a CHS:

```
    cylinder = LBA / (heads_per_cylinder * sectors_per_track)
        temp = LBA % (heads_per_cylinder * sectors_per_track)
        head = temp / sectors_per_track
      sector = temp % sectors_per_track + 1
```

While most OS's compute disk addresses in an LBA scheme, an OS like DOS must convert that LBA to a CHS in order to call INT 13H.

Technically an INT 13H should follow this process when converting an L-CHS to a P-CHS:

1) convert the L-CHS to an LBA,
2) convert the LBA to a P-CHS,

If an LBA is required at the ATA interface, then this third step is needed:

3) convert the P-CHS to an LBA.

All of these conversions are done by normal arithmetic.

However, while this is the technically correct way to do things, certain short cuts can be taken. It is possible to convert an L-CHS directly to a P-CHS using bit a bit shifting algorithm. This combines steps 1 and 2. And, if the ATA device being used supports LBA, steps 2 and 3 are not needed. The LBA value produced in step 1 is the same as the LBA value produced in step 3.

AN EXAMPLE

Lets look at an example. Lets say that the L-CHS is 1000 cylinders 10 heads and 50 sectors, the P-CHS is 2000 cylinders, 5 heads and 50 sectors. Lets say we want to access the sector at L-CHS 2,4,3.

* step 1 converts the L-CHS to an LBA,

```
     lba = 1202 = ( ( 2 * 10 + 4 ) * 50 ) + 3 - 1
```

* step 2 converts the LBA to the P-CHS,

```
     cylinder =   4 = ( 1202 / ( 5 * 50 )
         temp = 202 = ( 1202 % ( 5 * 50 ) )
         head =   4 = ( 202 / 50 )
       sector =   3 = ( 202 % 50 ) + 1
```

* step 3 converts the P-CHS to an LBA,

```
     lba = 1202 = ( ( 4 * 5 + 4 ) * 50 ) + 3 - 1
```

Most BIOS (or OS) software is not going to do all of this to convert an address. Most will use some other algorithm. There are many such algorithms.

BIT SHIFTING INSTEAD

If the L-CHS is produced from the P-CHS by 1) dividing the P-CHS cylinders by N, and 2) multiplying the P-CHS heads by N, where N is 2, 4, 8, ..., then this bit shifting algorithm can be used and N becomes a bit shift value. This is the most common way to make the P-CHS geometry of a >528MB drive fit the INT 13H L-CHS rules. Plus this algorithm maintains the same sector ordering as the more complex algorithm above. Note the following:

```
     Lcylinder = L-CHS cylinder being accessed
         Lhead = L-CHS head being accessed
       Lsector = L-CHS sector being accessed

     Pcylinder = the P-CHS cylinder being accessed
         Phead = the P-CHS head being accessed
       Psector = P-CHS sector being accessed

           NPH = is the number of heads in the P-CHS
             N = 2, 4, 8, ..., the bit shift value
```

The algorithm, which can be implemented using bit shifting instead of multiply and divide operations is:

```
     Pcylinder = ( Lcylinder * N ) + ( Lhead / NPH );
         Phead = ( Lhead % NPH );
       Psector = Lsector;
```

A BIT SHIFTING EXAMPLE

Lets apply this to our example above (L-CHS = 1000,10,50 and P-CHS = 2000, 5, 50) and access the same sector at at L-CHS 2,4,3.

```
     Pcylinder = 4 = ( 2 * 2 ) + ( 4 / 5 )
         Phead = 4 = ( 4 % 5 )
       Psector = 3 = 3
```

As you can see, this produces the same P-CHS as the more complex method above.

SO WHAT IS THE PROBLEM?

The basic problem is that there is no requirement that a CHS translating BIOS followed these rules. There are many other algorithms that can be implemented to perform a similar function. Today, there are at least two popular implementions: the Phoenix implementation (described above) and the non-Phoenix implementations.

SO WHY IS THIS A PROBLEM IF IT IS HIDDEN INSIDE THE BIOS?

Because a protected mode OS that does not want to use INT 13H must implement the same CHS translation algorithm. If it doesn't, your data gets scrambled.

WHY USE CHS AT ALL?

In the perfect world of tomorrow, maybe only LBA will be used. But today we are faced with the following problems:

* Some drives >528MB don't implement LBA.

* Some drives are optimized for CHS and may have lower performance when given commands in LBA mode. Don't forget that LBA is something new for the ATA disk designers who have worked very hard for many years to optimize CHS address handling. And not all drive designs require the use of LBA internally.

* The L-CHS to LBA conversion is more complex and slower than the bit shifting L-CHS to P-CHS conversion.

* DOS, FDISK and the MBR are still CHS based -- they use the CHS returned by INT 13H AH=08H. Any OS that can be installed on the same disk with DOS must understand CHS addressing.

* The BIOS boot processing and loading of the first OS kernel code is done in CHS mode -- the CHS returned by INT 13H AH=08H is used.

* Microsoft has said that their OS's will not use any disk capacity that can not also be accessed by INT 13H AH=0xH.

These are difficult problems to overcome in today's industry environment. The result: chaos.

DANGER TO YOUR DATA!

See the description of BIOS Type 7 below to understand why a WD EIDE BIOS is so dangerous to your data.

The BIOS Types

I assume the following:

a) All BIOS INT 13H support has been installed by the time the OS starts its boot processing. I'm don't plan to cover what could happen to INT 13H once the OS starts loading its own device drivers.

b) Drives supported by INT 13H are numbered sequentially starting with drive number 80H (80H-FFH are hard drives, 00-7FH are floppy drives).

And remember, any time a P-CHS exists it may or may not account for the CE Cylinder properly.

I have identified the following types of BIOS INT 13H support as seen by an OS during its boot time hardware configuration determination:

BIOS Type 1

Origin: Original IBM PC/XT.

BIOS call support: INT 13H AH=0xH and FDPT for BIOS drives 80H and 81H. There is no CHS translation. INT 13H AH=08H returns the P-CHS. The FDPT should contain the same P-CHS.

Description: Supports up to 528MB from a table of drive descriptions in BIOS ROM. No support for >1024 cylinders or drives >528MB or LBA.

Support issues: For >1024 cylinders or >528MB support, either an option ROM with an INT 13H replacement (see BIOS types 4-7) -or- a software driver (see BIOS type 8) must be added to the system.

BIOS Type 2

Origin: Unknown, but first appeared on systems having BIOS drive type table entries defining >1024 cylinders. Rumored to have originated at the request of Novell or SCO.

BIOS call support: INT 13H AH=0xH and FDPT for BIOS drives 80H and 81H. INT 13H AH=08H should return a L-CHS with the cylinder value limited to 1024. Beware, many BIOS perform a logical AND on the cylinder value. A correct BIOS will limit the cylinder value as follows:

```
cylinder = cylinder > 1024 ? 1024 : cylinder;
```

An incorrect BIOS will limit the cylinder value as follows (this implementation turns a 540MB drive into a 12MB drive!):

```
cylinder = cylinder & 0x03ff;
```

The FDPT will return a P-CHS that has the full cylinder value.

Description: For BIOS drive numbers 80H and 81H, this BIOS type supports >1024 cylinders or >528MB without using a translated CHS in the FDPT. INT 13H AH=08H truncates cylinders to 1024 (beware of buggy implementations). The FDPT can show >1024 cylinders thereby allowing an OS to support drives >528MB. May convert the L-CHS or P-CHS directly to an LBA if the ATA device supports LBA.

Support issues: Actual support of >1024 cylinders is OS specific -- some OS's may be able to place OS specific partitions spanning or beyond cylinder 1024. Usually all OS boot code must be within first 1024 cylinders. The FDISK program of an OS that supports such partitions uses an OS specific partition table entry format to identify these paritions. There does not appear to be a standard (de facto or otherwise) for this unusual partition table entry.

Apparently one method is to place -1 into the CHS fields and use the LBA fields to describe the location of the partition. This DOES NOT require the drive to support LBA addressing. Using an LBA in the partition table entry is just a trick to get around the CHS limits in the partition table entry. It is unclear if such a partition table entry will be ignored by an OS that does not understand what it is. For an OS that does not support such partitions, either an option ROM with an INT 13H replacement (see BIOS types 4-7) -or- a software driver (see BIOS type 8) must be added to the system.

Note: OS/2 can place HPFS partitions and Linux can place Linux partitions beyond or spanning cylinder 1024. (Anyone know of other systems that can do the same?)

BIOS Type 3

Origin: Unknown, but first appeared on systems having BIOS drive type table entires defining >1024 cylinders. Rumored to have originated at the request of Novell or SCO.

BIOS call support: INT 13H AH=0xH and FDPT for BIOS drives 80H and 81H. INT 13H AH=08H can return an L-CHS with more than 1024 cylinders.

Description: This BIOS is like type 2 above but it allows up to 4096 cylinders (12 cylinder bits). It does this in the INT 13H AH=0xH calls by placing two most significant cylinder bits (bits 11 and 10) into the upper two bits of the head number (bits 7 and 6).

Support issues: Identification of such a BIOS is difficult. As long as the drive(s) supported by this type of BIOS have <1024 cylinders this BIOS looks like a type 2 BIOS because INT 13H AH=08H should return zero data in bits 7 and 6 of the head information. If INT 13H AH=08H returns non zero data in bits 7 and 6 of the head information, perhaps it can be assumed that this is a type 3 BIOS. For more normal support of >1024 cylinders or >528MB, either an option ROM with an INT 13H replacement (see BIOS types 4-7) -or- a software driver (see BIOS type 8) must be added to the system.

Note: Apparently this BIOS type is no longer produced by any BIOS vendor.

BIOS Type 4

Origin: Compaq. Probably first appeared in systems with ESDI drives having >1024 cylinders.

BIOS call support: INT 13H AH=0xH and EDPT for BIOS drives 80H and 81H. If the drive has <1024 cylinders, INT 13H AH=08H returns the P-CHS and a FDPT is built. If the drive has >1024 cylinders, INT 13H AH=08H returns an L-CHS and an EDPT is built.

Description: Looks like a type 2 BIOS when an FDPT is built. Uses CHS translation when an EDPT is used. May convert the L-CHS directly to an LBA if the ATA device supports LBA.

Support issues: This BIOS type may support up to four drives with a EDPT (or FDPT) for BIOS drive numbers 82H and 83H located in memory following the EDPT (or FDPT) for drive 80H. Different CHS translation algorithms may be used by the BIOS and an OS.

BIOS Type 5

Origin: The IBM/Microsoft BIOS Extensions document. For many years this document was marked "confidential" so it did not get wide spread distribution.

BIOS call support: INT 13H AH=0xH, AH=4xH and EDPT for BIOS drives 80H and 81H. INT 13H AH=08H returns an L-CHS. INT 13H AH=41H and AH=48H should be used to get P-CHS configuration. The FDPT/EDPT should not be used. In some implementations the FDPT/EDPT may not exist.

Description: A BIOS that supports very large drives (>1024 cylinders, >528MB, actually >8GB), and supports the INT 13H AH=4xH read/write functions. The AH=4xH calls use LBA addressing and support drives with up to 2^64 sectors. These calls do NOT require that a drive support LBA at the drive interface. INT 13H AH=48H describes the L-CHS used at the INT 13 interface and the P-CHS or LBA used at the drive interface. This BIOS supports the INT 13 AH=0xH calls the same as a BIOS type 4.

Support issues: While the INT 13H AH=4xH calls are well defined, they are not implemented in many systems shipping today. Currently undefined is how such a BIOS should respond to INT 13H AH=08H calls for a drive that is >8GB. Different CHS translation algorithms may be used by the BIOS and an OS.

Note: Support of LBA at the drive interface may be automatic or may be under user control via a BIOS setup option. Use of LBA at the drive interface does not change the operation of the INT 13 interface.

BIOS Type 6

Origin: The Phoenix Enhanced Disk Drive Specification.

BIOS call support: INT 13H AH=0xH, AH=4xH and EDPT for BIOS drives 80H and 81H. INT 13H AH=08H returns an L-CHS. INT 13H AH=41H and AH=48H should be used to get P-CHS configuration. INT 13H AH=48H returns the address of the Phoenix defined "FDPT Extension" table.

Description: A BIOS that supports very large drives (>1024 cylinders, >528MB, actually >8GB), and supports the INT 13H AH=4xH read/write functions. The AH=4xH calls use LBA addressing and support drives with up to 2^64 sectors. These calls do NOT require that a drive support LBA at the drive interface. INT 13H AH=48H describes the L-CHS used at the INT 13 interface and the P-CHS or LBA used at the drive interface. This BIOS supports the INT 13 AH=0xH calls the same as a BIOS type 4. The INT 13H AH=48H call returns additional information such as host adapter addresses, DMA support, LBA support, etc, in the Phoenix defined "FDPT Extension" table.

Phoenix says this this BIOS need not support the INT 13H AH=4xH read/write calls but this BIOS is really an extension/enhancement of the original IBM/MS BIOS so most implementations will probably support the full set of INT 13H AH=4xH calls.

Support issues: Currently undefined is how such a BIOS should respond to INT 13H AH=08H calls for a drive that is >8GB. Different CHS translation algorithms may be used by the BIOS and an OS.

Note: Support of LBA at the drive interface may be automatic or may be under user control via a BIOS setup option. Use of LBA at the drive interface does not change the operation of the INT 13 interface.

BIOS Type 7

Origin: Described in the Western Digital Enhanced IDE Implementation Guide.

BIOS call support: INT 13H AH=0xH and FDPT or EDPT for BIOS drives 80H and 81H. An EDPT with a L-CHS of 16 heads and 63 sectors is built when "LBA mode" is enabled. An FDPT is built when "LBA mode" is disabled.

Description: Supports >1024 cylinders or >528MB using a EDPT with a translated CHS *** BUT ONLY IF *** the user requests "LBA mode" in the BIOS setup *** AND *** the drive supports LBA. As long as "LBA mode" is enabled, CHS translation is enabled using a L-CHS with <=1024 cylinders, 16, 32, 64, ..., heads and 63 sectors. Disk read/write commands are issued in LBA mode at the ATA interface but other commands are issued in P-CHS mode. Because the L-CHS is determined by table lookup based on total drive capacity and not by a multiply/divide of the P-CHS cylinder and head values, it may not be possible to use the simple (and faster) bit shifting L-CHS to P-CHS algorithms.

When "LBA mode" is disabled, this BIOS looks like a BIOS type 2 with an FDPT. The L-CHS used is taken either from the BIOS drive type table or from the device's Identify Device data. This L-CHS can be very different from the L-CHS returned when "LBA mode" is enabled.

This BIOS may support FDPT/EDPT for up to four drives in the same manner as described in BIOS type 4.

The basic problem with this BIOS is that the CHS returned by INT 13H AH=08H changes because of a change in the "LBA mode" setting in the BIOS setup. This should not happen. This use or non-use of LBA at the ATA interface should have no effect on the CHS returned by INT 13H AH=08H. This is the only BIOS type know to have this problem.

Support issues: If the user changes the "LBA mode" setting in BIOS setup, INT 13H AH=08H and the FDPT/EDPT change which may cause *** DATA CORRUPTION ***. The user should be warned to not change the "LBA mode" setting in BIOS setup once the drive has been partitioned and software installed. Different CHS translation algorithms may be used by the BIOS and an OS.

BIOS Type 8

Origin: Unknown. Perhaps Ontrack's Disk Manager was the first of these software drivers. Another example of such a driver is Micro House's EZ Drive.

BIOS call support: Unknown (anyone care to help out here?). Mostly likely only INT 13H AH=0xH are support. Probably a

FDPT or EDPT exists for drives 80H and 81H.

! Description: A software driver that "hides" in the MBR such that it is loaded into system memory before any OS boot processing starts. These drivers can have up to three parts: a part that hides in the MBR, a part that hides in the remaining sectors of cylinder 0, head 0, and an OS device driver. The part in the MBR loads the second part of the driver from cylinder 0 head 0. The second part provides a replacement for INT 13H that enables CHS translation for at least the boot drive. Usually the boot drive is defined in CMOS setup as a type 1 or 2 (5MB or 10MB drive). Once the second part of the driver is loaded, this definition is changed to describe the true capacity of the drive and INT 13H is replaced by the driver's version of INT 13H that does CHS translation. In some cases the third part, an OS specific device driver, must be loaded to enable CHS translation for devices other than the boot device.

! I don't know the details of how these drivers respond to INT 13H AH=08H or how they set up drive parameter tables (anyone care to help out here?). Some of these drivers convert the L-CHS to an LBA, then they add a small number to the LBA and finally they convert the LBA to a P-CHS. This in effect skips over some sectors at the front of the disk.

Support issues: Several identified -- Some OS installation programs will remove or overlay these drivers; some of these drivers do not perform CHS translation using the same algorithms used by the other BIOS types; special OS device drivers may be required in order to use these software drivers For example, under MS Windows the standard FastDisk driver (the 32-bit disk access driver) must be replaced by a driver that understands the Ontrack, Micro House, etc, version of INT 13H. Different CHS translation algorithms may be used by the driver and an OS.

! The hard disk vendors have been shipping these drivers with their drives over 528MB during the last year and they have been ignoring the statements of Microsoft and IBM that these drivers would not be supported in future OS's. Now it appears that both Microsoft and IBM are in a panic trying to figure out how to support some of these drivers in WinNT, Win95 and OS/2. It is unclear what the outcome of this will be at this time.

! NOTE: THIS IS NOT A PRODUCT ENDORSEMENT! An alternate solution for an older ISA system is one of the BIOS replacement cards. This cards have a BIOS option ROM. AMI makes such a card called the "Disk Extender". This card replaces the motherboard's INT 13H BIOS with a INT 13H BIOS that does some form of CHS translation. Another solution for older VL-Bus systems is an ATA-2 (EIDE) type host adapter card that provides a option ROM with an INT 13H replacement.

BIOS Type 9

Origin: SCSI host adapters.

BIOS call support: Probably INT 13H AH=0xH and FDPT for BIOS drives 80H and 81H, perhaps INT 13H AH=4xH.

Description: Most SCSI host adapters contain an option ROM that enables INT 13 support for the attached SCSI hard drives.

It is possible to have more than one SCSI host adapter, each with its own option ROM. The CHS used at the INT 13H interface is converted to the LBA that is used in the SCSI commands. INT 13H AH=08H returns a CHS. This CHS will have <=1024 cylinders, <=256 heads and <=63 sectors. The FDPT probably will exist for SCSI drives with BIOS drive numbers of 80H and 81H and probably indicates the same CHS as that returned by INT 13H AH=08H. Even though the CHS used at the INT 13H interface looks like a translated CHS, there is no need to use a EDPT since there is no CHS-to-CHS translation used. Other BIOS calls (most likely host adapter specific) must be used to determine other information about the host adapter or the drives.

The INT 13H AH=4xH calls can be used to get beyond 8GB but since there is little support for these calls in today's OS's, there are probably few SCSI host adapters that support these newer INT 13H calls.

Support issues: Some SCSI host adapters will not install their option ROM if there are two INT 13H devices previously installed by another INT 13H BIOS (for example, two MFM/RLL/ESDI/ATA drives). Other SCSI adapters will install their option ROM and use BIOS drive numbers greater than 81H. Some older OS's don't understand or use BIOS drive numbers greater than 81H. SCSI adapters are currently faced with the >8GB drive problem.

BIOS Type 10

Origin: A european system vendor.

BIOS call support: INT 13H AH=0xH and FDPT for BIOS drives 80H and 81H.

Description: This BIOS supports drives >528MB but it does not support CHS translation. It supports only ATA drives with LBA capability. INT 13H AH=08H returns an L-CHS. The L-CHS is converted directly to an LBA. The BIOS sets the ATA drive to a P-CHS of 16 heads and 63 sectors using the Initialize Drive Parameters command but it does not use this P-CHS at the ATA interface.

! Support issues: OS/2 will probably work with this BIOS as long as the drive's power on default P-CHS mode uses 16 heads and 63 sectors. Because there is no EDPT, OS/2 uses the ATA Identify Device power on default P-CHS, described in Identify Device words 1, 3 and 6 as the current P-CHS for the drive. However, this may not represent the correct P-CHS. A newer drive will have the its current P-CHS information in Identify Device words 53-58 but for some reason OS/2 does not use this information.

How it Works -- Partition Tables

Version 1b

by Hale Landis (landis@sugs.tware.com)

THE "HOW IT WORKS" SERIES

This is one of several How It Works documents. The series currently includes the following:

* How It Works -- CHS Translation
* How It Works -- Master Boot Record
* How It Works -- DOS Floppy Boot Sector
* How It Works -- OS2 Boot Sector
* How It Works -- Partition Tables

PARTITION SECTOR/RECORD/TABLE BASICS

FDISK creates all partition records (sectors). The primary purpose of a partition record is to hold a partition table. The rules for how FDISK works are unwritten but so far most FDISK programs (DOS, OS/2, WinNT, etc) seem to follow the same basic idea.

First, all partition table records (sectors) have the same format. This includes the partition table record at cylinder 0, head 0, sector 1 -- what is known as the Master Boot Record (MBR). The last 66 bytes of a partition table record contain a partition table and a 2 byte signature. The first 446 bytes of these sectors usually contain a program but only the program in the MBR is ever executed (so extended partition table records could contain something other than a program in the first 466 bytes). See "How It Works -- The Master Boot Record".

Second, extended partitions are "nested" inside one another and extended partition table records form a "linked list". I will attempt to show this in a diagram below.

PARTITION TABLE ENTRY FORMAT

Each partition table entry is 16 bytes and contains things like the start and end location of a partition in CHS, the start in LBA, the size in sectors, the partition "type" and the "active" flag. Warning: older versions of FDISK may compute incorrect LBA or size values. And note: When your computer boots itself, only the CHS fields of the partition table entries are used (another reason LBA doesn't solve the >528MB problem). The CHS fields in the partition tables are in L-CHS format -- see "How It Works -- CHS Translation".

There is no central clearing house to assign the codes used in the one byte "type" field. But codes are assigned (or used) to define most every type of file system that anyone has ever implemented on the x86 PC: 12-bit FAT, 16-bit FAT, HPFS, NTFS, etc. Plus, an extended partition also has a unique type code.

Note: I know of no complete list of all the type codes that have been used to date. However, I try to include such a list in a future version of this document.

The 16 bytes of a partition table entry are used as follows:

```
    +--- Bit 7 is the active partition flag, bits 6-0 are zero.
    |
    |    +--- Starting CHS in INT 13 call format.
    |    |
    |    |      +--- Partition type byte.
    |    |      |
    |    |      |    +--- Ending CHS in INT 13 call format.
    |    |      |    |
```

```
   |    |      |    |        +-- Starting LBA.
   |    |      |    |        |
   |    |      |    |        |       +-- Size in sectors.
   |    |      |    |        |       |
   v <--+--->  v <--+-->     v       v

  0  1  2  3  4  5  6  7  8 9 A B  C D E F
  DH DL CH CL TB DL CH CL LBA..... SIZE....

  80 01 01 00 06 0e be 94 3e000000 0c610900  1st entry

  00 00 81 95 05 0e fe 7d 4a610900 724e0300  2nd entry

  00 00 00 00 00 00 00 00 00000000 00000000  3rd entry

  00 00 00 00 00 00 00 00 00000000 00000000  4th entry
```

Bytes 0-3 are used by the small program in the Master Boot Record to read the first sector of an active partition into memory. The DH, DL, CH and CL above show which x86 register is loaded when the MBR program calls INT 13H AH=02H to read the active partition's boot sector. See "How It Works -- Master Boot Record".

These entries define the following partitions:

1) The first partition, a primary partition DOS FAT, starts at CHS 0H,1H,1H (LBA 3EH) and ends at CHS 294H,EH,3EH with a size of 9610CH sectors.

2) The second partition, an extended partition, starts at CHS 295H,0H,1H (LBA 9614AH) and ends at CHS 37DH,EH,3EH with a size of 34E72H sectors.

3) The third and fourth table entries are unused.

PARTITION TABLE RULES

Keep in mind that there are NO written rules and NO industry standards on how FDISK should work but here are some basic rules that seem to be followed by most versions of FDISK:

1) In the MBR there can be 0-4 "primary" partitions, OR, 0-3 primary partitions and 0-1 extended partition entry.

2) In an extended partition there can be 0-1 "secondary" partition entries and 0-1 extended partition entries.

3) Only 1 primary partition in the MBR can be marked "active" at any given time.

4) In most versions of FDISK, the first sector of a partition will be aligned such that it is at head 0, sector 1 of a cylinder. This means that there may be unused sectors on the track(s) prior to the first sector of a partition and that there may be unused sectors following a partition table sector.

   For example, most new versions of FDISK start the first partition (primary or extended) at cylinder 0, head 1, sector 0. This leaves the sectors at cylinder 0, head 0, sectors 2...n as unused sectors. This same layout may be seen on the first track of an extended partition. See example 2 below.

Also note that software drivers like Ontrack's Disk Manager depend on these unused sectors because these drivers will "hide" their code there (in cylinder 0, head 0, sectors 2...n). This is also a good place for boot sector virus programs to hang out.

5) The partition table entries (slots) can be used in any order. Some versions of FDISK fill the table from the bottom up and some versions of FDISK fill the table from the top down. Deleting a partition can leave an unused entry (slot) in the middle of a table.

6) And then there is the "hack" that some newer OS's (OS/2 and Linux) use in order to place a partition spanning or passed cylinder 1024 on a system that does not have a CHS translating BIOS. These systems create a partition table entry with the partition's starting and ending CHS information set to all FFH. The starting and ending LBA information is used to describe the location of the partition. The LBA can be converted back to a CHS -- most likely a CHS with more than 1024 cylinders. Since such a CHS can't be used by the system BIOS, these partitions can not be booted or accessed until the OS's kernel and hard disk device drivers are loaded. It is not known if the systems using this "hack" follow the same rules for the creation of these type of partitions.

There are NO written rules as to how an OS scans the partition table entries so each OS can have a different method. For DOS, this means that different versions could assign different drive letters to the same FAT file system partitions.

PARTITION NESTING

What do I mean when I say the partitions are "nested" within each other? Lets look at this example:

```
     M = Master Boot Record (and any unused sectors
         on the same track)
     E = Extended partition record (and any unused sectors
         on the same track)
   pri = a primary partition (first sector is a "boot" sector)
   sec = a secondary partition (first sector is a "boot" sector)


  |<-----------------the entire disk-------------->|
  |                                                |
  |M<pri>                                          |
  |                                                |
  |      E<sec><---rest of 1st ext part----------->|
  |                                                |
  |            E<sec><---rest of 2nd ext part----->|
```

The first extended partition is described in the MBR and it occupies the entire disk following the primary partition. The second extended partition is described in the first extended partition record and it occupies the entire disk following the first secondary partition.

PARTITION TABLE LINKING

What do I mean when I say the partition records (tables) form a "linked" list? This means that the MBR has an entry that describes (points to) the first extended partition, the first extended partition table has an entry that describes (points to) the second extended partition table, and so on. There is, in theory, no limited to out long this linked list is. When you ask FDISK to show the DOS "logical drives" it scans the linked list looking for all of the DOS FAT type partitions that may exist. Remember that in an extended partition table, only two entries of the four can be used (rule 2 above).

And one more thing... Within a partition, the layout of the file system data varies greatly. However, the first sector of a partition is expected to be a "boot" sector. A DOS FAT file system has: a boot sector, first FAT sectors, second FAT sectors, root directory sectors and finally the file data area. See "How It Works -- OS2 Boot Sector".

EXAMPLE 1

A disk containing four DOS FAT partitions (C, D, E and F):

```
  |<----------------------the entire disk------------------->|
  |                                                          |
  |M<---C:--->                                               |
  |                                                          |
  |           E<---D:---><-rest of 1st ext part------------>|
  |                                                          |
  |                      E<---E:---><-rest of 2nd ext part->|
  |                                                          |
  |                                 E<---------F:---------->|
```

EXAMPLE 2

So here is an example of a disk with two primary partitions, one DOS FAT and one OS/2 HPFS, plus an extended partition with another DOS FAT:

```
  |<-------------------the entire disk------------------>|
  |                                                      |
  |M<pri 1 - DOS FAT>                                    |
  |                                                      |
  |                    <pri 2 - OS/2 HPFS>               |
  |                                                      |
  |                                     E<sec - DOS FAT>|
```

Or in more detail ('n' is the highest cylinder, head or sector number number allowed in the indicated field of the CHS)...

```
            +-------------------------------------+
 CHS=0,0,1  | Master Boot Record containing       |
            | partition table search program and  |
            | a partition table                   |
            | +---------------------------------+ |
            | | DOS FAT partition description   | | points to CHS=0,1,1
            | +---------------------------------+ | points to CHS=a
            | | OS/2 HPFS partition description | |
            | +---------------------------------+ |
            | | unused table entry              | |
            | +---------------------------------+ |
            | | extended partition entry        | | points to CHS=b
```

```
             | +--------------------------------+ |
             +------------------------------------+
CHS=0,0,2    | the rest of "track 0" -- this is   | :
to           | where the software drivers such as | : normally
CHS=0,0,n    | Ontrack's Disk Manager or Micro    | : unused
             | House's EZ Drive are located.      | :
             +------------------------------------+
CHS=0,1,1    | Boot sector for the DOS FAT        | :
             | partition                          | : a DOS FAT
             +------------------------------------+ : file
CHS=0,1,2    | rest of the DOS FAT partition      | : system
to           | (FAT table, root directory and     | :
CHS=x-1,n,n  | user data area)                    | :
             +------------------------------------+
CHS=x,0,1    | Boot sector for the OS/2 HPFS      | :
             | file system partition              | : an OS/2
             +------------------------------------+ : HPFS file
CHS=x,0,2    | rest of the OS/2 HPFS file system  | : system
to           | partition                          | :
CHS=y-1,n,n  |                                    | :
             +------------------------------------+
CHS=y,0,1    | Partition record for the extended  |
             | partition containing a partition   |
             | record program (never executed) and |
             | a partition table                  |
             | +--------------------------------+ |
             | | DOS FAT partition description  | | points to CHS=b+1
             | +--------------------------------+ |
             | | unused table entry             | |
             | +--------------------------------+ |
             | | unused table entry             | |
             | +--------------------------------+ |
             | | unused table entry             | |
             | +--------------------------------+ |
             +------------------------------------+
CHS=y,0,2    | the rest of the first track of the | : normally
to           | extended partition                 | : unused
CHS=y,0,n    |                                    | :
             +------------------------------------+
CHS=y,1,1    | Boot sector for the DOS FAT        | :
             | partition                          | : a DOS FAT
             +------------------------------------+ : file
CHS=y,1,2    | rest of the DOS FAT partition      | : system
to           | (FAT table, root directory and     | :
CHS=n,n,n    | user data area)                    | :
             +------------------------------------+
```

EXAMPLE 3

Here is a partition record from an extended partition (the first sector of an extended partition). Note that it contains no program code. It contains only the partition table and the signature data.

```
OFFSET 0 1 2 3  4 5 6 7  8 9 A B  C D E F  *0123456789ABCDEF*
000000 00000000 00000000 00000000 00000000 *................*
000010 TO 0001af SAME AS ABOVE
0001b0 00000000 00000000 00000000 00000001 *................*
0001c0 8195060e fe7d3e00 0000344e 03000000 *.....}>...4N....*
0001d0 00000000 00000000 00000000 00000000 *................*
0001e0 00000000 00000000 00000000 00000000 *................*
0001f0 00000000 00000000 00000000 000055aa *..............U.*
```

NOTES

Thanks to yue@heron.Stanford.EDU (Kenneth C. Yue) for pointing out that in V0 of this document I did not properly describe the unused sectors normally found around the partition table sectors.

How It Works -- Master Boot Record

Version 1a

by Hale Landis (landis@sugs.tware.com)

THE "HOW IT WORKS" SERIES

This is one of several How It Works documents. The series currently includes the following:

* How It Works -- CHS Translation
* How It Works -- Master Boot Record
* How It Works -- DOS Floppy Boot Sector
* How It Works -- OS2 Boot Sector
* How It Works -- Partition Tables

MASTER BOOT RECORD

This article is a disassembly of a Master Boot Record (MBR). The MBR is the sector at cylinder 0, head 0, sector 1 of a hard disk. An MBR is created by the FDISK program. The FDISK program of all operating systems must create a functionally similar MBR. The MBR is first of what could be many partition sectors, each one containing a four entry partition table.

At the completion of your system's Power On Self Test (POST), INT 19 is called. Usually INT 19 tries to read a boot sector from the first floppy drive. If a boot sector is found on the floppy disk, the that boot sector is read into memory at location 0000:7C00 and INT 19 jumps to memory location 0000:7C00. However, if no boot sector is found on the first floppy drive, INT 19 tries to read the MBR from the first hard drive. If an MBR is found it is read into memory at location 0000:7c00 and INT 19 jumps to memory location 0000:7c00. The small program in the MBR will attempt to locate an active (bootable) partition in its partition table. If such a partition is found, the boot sector of that partition is read into memory at location 0000:7C00 and the MBR program jumps to memory location 0000:7C00. Each operating system has its own boot sector format. The small program in the boot sector must locate the first part of the operating system's kernel loader program (or perhaps the kernel itself or perhaps a "boot manager program") and read that into memory.

INT 19 is also called when the CTRL-ALT-DEL keys are used. On most systems, CTRL-ALT-DEL causes an short version of the POST to be executed before INT 19 is called.

=====

Where stuff is:

    The MBR program code starts at offset 0000.
    The MBR messages start at offset 008b.

The partition table starts at offset 00be.
The signature is at offset 00fe.

Here is a summary of what this thing does:

If an active partition is found, that partition's boot record is read into 0000:7c00 and the MBR code jumps to 0000:7c00 with SI pointing to the partition table entry that describes the partition being booted. The boot record program uses this data to determine the drive being booted from and the location of the partition on the disk.

If no active partition table enty is found, ROM BASIC is entered via INT 18. All other errors cause a system hang, see label HANG.

NOTES (VERY IMPORTANT):

1) The first byte of an active partition table entry is 80. This byte is loaded into the DL register before INT 13 is called to read the boot sector. When INT 13 is called, DL is the BIOS device number. Because of this, the boot sector read by this MBR program can only be read from BIOS device number 80 (the first hard disk). This is one of the reasons why it is usually not possible to boot from any other hard disk.

2) The MBR program uses the CHS based INT 13H AH=02H call to read the boot sector of the active partition. The location of the active partition's boot sector is in the partition table entry in CHS format. If the drive is >528MB, this CHS must be a translated CHS (or L-CHS, see my BIOS TYPES document). No addresses in LBA form are used (another reason why LBA doesn't solve the >528MB problem).

=====

Here is the entire MBR record (hex dump and ascii).

```
OFFSET 0 1 2 3  4 5 6 7  8 9 A B  C D E F  *0123456789ABCDEF*
000000 fa33c08e d0bc007c 8bf45007 501ffbfc *.3.....|..P.P...*
000010 bf0006b9 0001f2a5 ea1d0600 00bebe07 *................*
000020 b304803c 80740e80 3c00751c 83c610fe *...<.t..<.u.....*
000030 cb75efcd 188b148b 4c028bee 83c610fe *.u......L.......*
000040 cb741a80 3c0074f4 be8b06ac 3c00740b *.t..<.t.....<.t.*
000050 56bb0700 b40ecd10 5eebf0eb febf0500 *V.......^.......*
000060 bb007cb8 010257cd 135f730c 33c0cd13 *..|...W.._s.3...*
000070 4f75edbe a306ebd3 bec206bf fe7d813d *Ou...........}.=*
000080 55aa75c7 8bf5ea00 7c000049 6e76616c *U.u.....|..Inval*
000090 69642070 61727469 74696f6e 20746162 *id partition tab*
0000a0 6c650045 72726f72 206c6f61 64696e67 *le.Error loading*
0000b0 206f7065 72617469 6e672073 79737465 * operating syste*
0000c0 6d004d69 7373696e 67206f70 65726174 *m.Missing operat*
0000d0 696e6720 73797374 656d0000 00000000 *ing system......*
0000e0 00000000 00000000 00000000 00000000 *................*
0000f0 TO 0001af SAME AS ABOVE
0001b0 00000000 00000000 00000000 00008001 *................*
0001c0 0100060d fef83e00 00000678 0d000000 *......>....x....*
0001d0 00000000 00000000 00000000 00000000 *................*
0001e0 00000000 00000000 00000000 00000000 *................*
0001f0 00000000 00000000 00000000 000055aa *..............U.*
```

=====

Here is the disassembly of the MBR...

This sector is initially loaded into memory at 0000:7c00 but
it immediately relocates itself to 0000:0600.

```
                 BEGIN:                      NOW AT 0000:7C00, RELOCATE

0000:7C00 FA             CLI                    disable int's
0000:7C01 33C0           XOR     AX,AX          set stack seg to 0000
0000:7C03 8ED0           MOV     SS,AX
0000:7C05 BC007C         MOV     SP,7C00        set stack ptr to 7c00
0000:7C08 8BF4           MOV     SI,SP          SI now 7c00
0000:7C0A 50             PUSH    AX
0000:7C0B 07             POP     ES             ES now 0000:7c00
0000:7C0C 50             PUSH    AX
0000:7C0D 1F             POP     DS             DS now 0000:7c00
0000:7C0E FB             STI                    allow int's
0000:7C0F FC             CLD                    clear direction
0000:7C10 BF0006         MOV     DI,0600        DI now 0600
0000:7C13 B90001         MOV     CX,0100        move 256 words (512 bytes)
0000:7C16 F2             REPNZ                  move MBR from 0000:7c00
0000:7C17 A5             MOVSW                     to 0000:0600
0000:7C18 EA1D060000     JMP     0000:061D      jmp to NEW_LOCATION

        NEW_LOCATION:                        NOW AT 0000:0600

0000:061D BEBE07       MOV     SI,07BE          point to first table entry
0000:0620 B304         MOV     BL,04            there are 4 table entries

        SEARCH_LOOP1:                        SEARCH FOR AN ACTIVE ENTRY

0000:0622 803C80       CMP     BYTE PTR [SI],80  is this the active entry?
0000:0625 740E         JZ      FOUND_ACTIVE      yes
0000:0627 803C00       CMP     BYTE PTR [SI],00  is this an inactive entry?
0000:062A 751C         JNZ     NOT_ACTIVE        no
0000:062C 83C610       ADD     SI,+10            incr table ptr by 16
0000:062F FECB         DEC     BL                decr count
0000:0631 75EF         JNZ     SEARCH_LOOP1      jmp if not end of table
0000:0633 CD18         INT     18                GO TO ROM BASIC

        FOUND_ACTIVE:                        FOUND THE ACTIVE ENTRY

0000:0635 8B14         MOV     DX,[SI]           set DH/DL for INT 13 call
0000:0637 8B4C02       MOV     CX,[SI+02]        set CH/CL for INT 13 call
0000:063A 8BEE         MOV     BP,SI             save table ptr

        SEARCH_LOOP2:                        MAKE SURE ONLY ONE ACTIVE ENTRY

0000:063C 83C610       ADD     SI,+10            incr table ptr by 16
0000:063F FECB         DEC     BL                decr count
0000:0641 741A         JZ      READ_BOOT         jmp if end of table
0000:0643 803C00       CMP     BYTE PTR [SI],00  is this an inactive entry?
0000:0646 74F4         JZ      SEARCH_LOOP2      yes

          NOT_ACTIVE:                        MORE THAN ONE ACTIVE ENTRY FOUND

0000:0648 BE8B06       MOV     SI,068B           display "Invld prttn tbl"

         DISPLAY_MSG:                        DISPLAY MESSAGE LOOP

0000:064B AC           LODSB                     get char of message
0000:064C 3C00         CMP     AL,00             end of message
0000:064E 740B         JZ      HANG              yes
0000:0650 56           PUSH    SI                save SI
0000:0651 BB0700       MOV     BX,0007           screen attributes
```

```
0000:0654 B40E         MOV     AH,0E              output 1 char of message
0000:0656 CD10         INT     10                    to the display
0000:0658 5E           POP     SI                 restore SI
0000:0659 EBF0         JMP     DISPLAY_MSG        do it again

            HANG:                              HANG THE SYSTEM LOOP

0000:065B EBFE         JMP     HANG               sit and stay!

       READ_BOOT:                              READ ACTIVE PARITION BOOT RECORD

0000:065D BF0500       MOV     DI,0005            INT 13 retry count

       INT13RTRY:                              INT 13 RETRY LOOP

0000:0660 BB007C       MOV     BX,7C00
0000:0663 B80102       MOV     AX,0201            read 1 sector
0000:0666 57           PUSH    DI                 save DI
0000:0667 CD13         INT     13                 read sector into 0000:7c00
0000:0669 5F           POP     DI                 restore DI
0000:066A 730C         JNB     INT13OK            jmp if no INT 13
0000:066C 33C0         XOR     AX,AX              call INT 13 and
0000:066E CD13         INT     13                    do disk reset
0000:0670 4F           DEC     DI                 decr DI
0000:0671 75ED         JNZ     INT13RTRY          if not zero, try again
0000:0673 BEA306       MOV     SI,06A3            display "Errr ldng systm"
0000:0676 EBD3         JMP     DISPLAY_MSG        jmp to display loop

         INT13OK:                              INT 13 ERROR

0000:0678 BEC206       MOV     SI,06C2               "missing op sys"
0000:067B BFFE7D       MOV     DI,7DFE               point to signature
0000:067E 813D55AA     CMP     WORD PTR [DI],AA55    is signature correct?
0000:0682 75C7         JNZ     DISPLAY_MSG           no
0000:0684 8BF5         MOV     SI,BP                 set SI
0000:0686 EA007C0000   JMP     0000:7C00             JUMP TO THE BOOT SECTOR
                                                        WITH SI POINTING TO
                                                        PART TABLE ENTRY

Messages here.

0000:0680 ........ ........ ......49 6e76616c *           Inval*
0000:0690 69642070 61727469 74696f6e 20746162 *id partition tab*
0000:06a0 6c650045 72726f72 206c6f61 64696e67 *le.Error loading*
0000:06b0 206f7065 72617469 6e672073 79737465 * operating syste*
0000:06c0 6d004d69 7373696e 67206f70 65726174 *m.Missing operat*
0000:06d0 696e6720 73797374 656d00.. ........ *ing system.     *

Data not used.

0000:06d0 ........ ........ ......00 00000000 *           .....*
0000:06e0 00000000 00000000 00000000 00000000 *................*
0000:06f0 00000000 00000000 00000000 00000000 *................*
0000:0700 00000000 00000000 00000000 00000000 *................*
0000:0710 00000000 00000000 00000000 00000000 *................*
0000:0720 00000000 00000000 00000000 00000000 *................*
0000:0730 00000000 00000000 00000000 00000000 *................*
0000:0740 00000000 00000000 00000000 00000000 *................*
0000:0750 00000000 00000000 00000000 00000000 *................*
0000:0760 00000000 00000000 00000000 00000000 *................*
0000:0770 00000000 00000000 00000000 00000000 *................*
0000:0780 00000000 00000000 00000000 00000000 *................*
0000:0790 00000000 00000000 00000000 00000000 *................*
0000:07a0 00000000 00000000 00000000 00000000 *................*
```

```
0000:07b0 00000000 00000000 00000000 0000.... *............    *
```

The partition table starts at 0000:07be. Each partition table entry is 16 bytes. This table defines a single primary partition which is also an active (bootable) partition.

```
0000:07b0 ........ ........ ........ ....8001 *            ....*
0000:07c0 0100060d fef83e00 00000678 0d000000 *......>....x....*
0000:07d0 00000000 00000000 00000000 00000000 *................*
0000:07e0 00000000 00000000 00000000 00000000 *................*
0000:07f0 00000000 00000000 00000000 0000.... *............    *
```

The last two bytes contain a 55AAH signature.

```
0000:07f0 ........ ........ ........ ....55aa *..............U.*
```

How It Works -- DOS Floppy Disk Boot Sector

Version 1a

by Hale Landis (landis@sugs.tware.com)

THE "HOW IT WORKS" SERIES

This is one of several How It Works documents. The series currently includes the following:

* How It Works -- CHS Translation
* How It Works -- Master Boot Record
* How It Works -- DOS Floppy Boot Sector
* How It Works -- OS2 Boot Sector
* How It Works -- Partition Tables

DOS FLOPPY DISK BOOT SECTOR

This article is a disassembly of a floppy disk boot sector for a DOS floppy. The boot sector of a floppy disk is located at cylinder 0, head 0, sector 1. This sector is created by a floppy disk formating program, such as the DOS FORMAT program. The boot sector of a FAT hard disk partition has a similar layout and function. Basically a bootable FAT hard disk partition looks like a big floppy during the early stages of the system's boot processing.

At the completion of your system's Power On Self Test (POST), INT 19 is called. Usually INT 19 tries to read a boot sector from the first floppy drive. If a boot sector is found on the floppy disk, the that boot sector is read into memory at location 0000:7C00 and INT 19 jumps to memory location 0000:7C00. However, if no boot sector is found on the first floppy drive, INT 19 tries to read the MBR from the first hard drive. If an MBR is found it is read into memory at location 0000:7c00 and INT 19 jumps to memory location 0000:7c00. The small program in the MBR will attempt to locate an active (bootable) partition in its partition table. If such a partition is found, the boot sector of that partition is read into memory at location 0000:7C00 and the MBR program jumps to memory location 0000:7C00. Each operating system has its own boot sector format. The small program in the boot sector must locate the first part of the operating system's kernel loader program (or perhaps the kernel itself or perhaps a "boot manager program") and read that into memory.

INT 19 is also called when the CTRL-ALT-DEL keys are used. On most systems, CTRL-ALT-DEL causes an short version of the POST to be executed before INT 19 is called.

=====

Where stuff is:

   The BIOS Parameter Block (BPB) starts at offset 0.
   The boot sector program starts at offset 3e.
   The messages issued by this program start at offset 19e.
   The DOS hidden file names start at offset 1e6.
   The boot sector signature is at offset 1fe.

Here is a summary of what this thing does:

1) Copy Diskette Parameter Table which is pointed to by INT 1E.
2) Alter the copy of the Diskette Parameter Table.
3) Alter INT 1E to point to altered Diskette Parameter Table.
4) Do INT 13 AH=00, disk reset call.
5) Compute sector address of root directory.
6) Read first sector of root directory into 0000:0500.
7) Confirm that first two directory entries are for IO.SYS and MSDOS.SYS.
8) Read first 3 sectors of IO.SYS into 0000:0700 (or 0070:0000).
9) Leave some information in the registers and jump to IO.SYS at 0070:0000.

NOTE:

   This program uses the CHS based INT 13H AH=02 to read the FAT root directory and to read the IO.SYS file. If the drive is >528MB, this CHS must be a translated CHS (or L-CHS, see my BIOS TYPES document). Except for internal computations no addresses in LBA form are used, another reason why LBA doesn't solve the >528MB problem.

=====

Here is the entire sector in hex and ascii.

```
OFFSET 0 1 2 3  4 5 6 7  8 9 A B  C D E F  *0123456789ABCDEF*
000000 eb3c904d 53444f53 352e3000 02010100 *.<.MSDOS5.0.....*
000010 02e00040 0bf00900 12000200 00000000 *...@............*
000020 00000000 0000295a 5418264e 4f204e41 *......)ZT.&NO NA*
000030 4d452020 20204641 54313220 2020fa33 *ME    FAT12   .3*
000040 c08ed0bc 007c1607 bb780036 c5371e56 *.....|...x.6.7.V*
000050 1653bf3e 7cb90b00 fcf3a406 1fc645fe *.S.>|.........E.*
000060 0f8b0e18 7c884df9 894702c7 073e7cfb *....|.M..G...>|.*
000070 cd137279 33c03906 137c7408 8b0e137c *..ry3.9..|t....|*
000080 890e207c a0107cf7 26167c03 061c7c13 *.. |..|.&.|...|.*
000090 161e7c03 060e7c83 d200a350 7c891652 *..|...|....P|..R*
0000a0 7ca3497c 89164b7c b82000f7 26117c8b *|.I|..K|. ..&.|.*
0000b0 1e0b7c03 c348f7f3 0106497c 83164b7c *..|..H....I|..K|*
0000c0 00bb0005 8b16527c a1507ce8 9200721d *......R|.P|...r.*
0000d0 b001e8ac 0072168b fbb90b00 bee67df3 *.....r........}.*
0000e0 a6750a8d 7f20b90b 00f3a674 18be9e7d *.u... .....t...}*
0000f0 e85f0033 c0cd165e 1f8f048f 4402cd19 *._.3...^....D...*
000100 585858eb e88b471a 48488a1e 0d7c32ff *XXX...G.HH...|2.*
000110 f7e30306 497c1316 4b7cbb00 07b90300 *....I|..K|......*
000120 505251e8 3a0072d8 b001e854 00595a58 *PRQ.:.r....T.YZX*
```

```
000130 72bb0501 0083d200 031e0b7c e2e28a2e *r..........|....*
000140 157c8a16 247c8b1e 497ca14b 7cea0000 *.|..$|..I|.K|...*
000150 7000ac0a c07429b4 0ebb0700 cd10ebf2 *p....t).........*
000160 3b16187c 7319f736 187cfec2 88164f7c *;..|s..6.|....O|*
000170 33d2f736 1a7c8816 257ca34d 7cf8c3f9 *3..6.|..%|.M|...*
000180 c3b4028b 164d7cb1 06d2e60a 364f7c8b *.....M|.....6O|.*
000190 ca86e98a 16247c8a 36257ccd 13c30d0a *.....$|.6%|.....*
0001a0 4e6f6e2d 53797374 656d2064 69736b20 *Non-System disk *
0001b0 6f722064 69736b20 6572726f 720d0a52 *or disk error..R*
0001c0 65706c61 63652061 6e642070 72657373 *eplace and press*
0001d0 20616e79 206b6579 20776865 6e207265 * any key when re*
0001e0 6164790d 0a00494f 20202020 20205359 *ady...IO      SY*
0001f0 534d5344 4f532020 20535953 000055aa *SMSDOS   SYS..U.*
```

=====

The first 62 bytes of a boot sector are known as the BIOS Parameter Block (BPB). Here is the layout of the BPB fields and the values they are assigned in this boot sector:

```
   db JMP instruction      at 7c00 size  2 = eb3c
   db NOP instruction         7c02       1   90
   db OEMname                 7c03       8   'MSDOS5.0'
   dw bytesPerSector          7c0b       2   0200
   db sectPerCluster          7c0d       1   01
   dw reservedSectors         7c0e       2   0001
   db numFAT                  7c10       1   02
   dw numRootDirEntries       7c11       2   00e0
   dw numSectors              7c13       2   0b40 (ignore numSectorsHuge)
   db mediaType               7c15       1   f0
   dw numFATsectors           7c16       2   0009
   dw sectorsPerTrack         7c18       2   0012
   dw numHeads                7c1a       2   0002
   dd numHiddenSectors        7c1c       4   00000000
   dd numSectorsHuge          7c20       4   00000000
   db driveNum                7c24       1   00
   db reserved                7c25       1   00
   db signature               7c26       1   29
   dd volumeID                7c27       4   5a541826
   db volumeLabel             7c2b      11   'NO NAME    '
   db fileSysType             7c36       8   'FAT12   '
```

=====

Here is the boot sector...

The first 3 bytes of the BPB are JMP and NOP instructions.

```
0000:7C00 EB3C        JMP     START
0000:7C02 90          NOP
```

Here is the rest of the BPB.

```
0000:7C00 ......4d 53444f53 352e3000 02010100 *   MSDOS5.0.....*
0000:7C10 02e00040 0bf00900 12000200 00000000 *...@............*
0000:7C20 00000000 0000295a 5418264e 4f204e41 *......)ZT.&NO NA*
0000:7C30 4d452020 20204641 54313220 2020.... *ME    FAT12     *
```

Now pay attention here...

The 11 bytes starting at 0000:7c3e are immediately overlaid by information copied from another part of memory. That information is the Diskette Parameter Table. This data is pointed to by INT 1E. This data is:

```
   7c3e = Step rate and head unload time.
   7c3f = Head load time and DMA mode flag.
   7c40 = Delay for motor turn off.
   7c41 = Bytes per sector.
   7c42 = Sectors per track.
   7c43 = Intersector gap length.
   7c44 = Data length.
   7c45 = Intersector gap length during format.
   7c46 = Format byte value.
   7c47 = Head settling time.
   7c48 = Delay until motor at normal speed.

   The 11 bytes starting at 0000:7c49 are also overlaid by the
   following data:

   7c49 - 7c4c = diskette sector address (as LBA)
                    of the data area.
   7c4d - 7c4e = cylinder number to read from.
   7c4f - 7c4f = sector number to read from.
   7c50 - 7c53 = diskette sector address (as LBA)
                    of the root directory.

            START:                        START OF BOOT SECTOR PROGRAM

0000:7C3E FA           CLI                          interrupts off
0000:7C3F 33C0         XOR     AX,AX                set AX to zero
0000:7C41 8ED0         MOV     SS,AX                SS is now zero
0000:7C43 BC007C       MOV     SP,7C00              SP is now 7c00
0000:7C46 16           PUSH    SS                   also set ES
0000:7C47 07           POP     ES                      to zero

                               The INT 1E vector is at 0000:0078.
                               Get the address that the vector points to
                               into the DS:SI registers.

0000:7C48 BB7800       MOV     BX,0078              BX is now 78
0000:7C4B 36           SS:
0000:7C4C C537         LDS     SI,[BX]              DS:SI is now [0:78]
0000:7C4E 1E           PUSH    DS                   save DS:SI --
0000:7C4F 56           PUSH    SI                      saves param tbl addr
0000:7C50 16           PUSH    SS                   save SS:BX --
0000:7C51 53           PUSH    BX                      saves INT 1E address

                               Move the diskette param table to 0000:7c3e.

0000:7C52 BF3E7C       MOV     DI,7C3E              DI is address of START
0000:7C55 B90B00       MOV     CX,000B              count is 11
0000:7C58 FC           CLD                          clear direction
0000:7C59 F3           REPZ                         move the diskette param
0000:7C5A A4           MOVSB                           table to 0000:7c3e
0000:7C5B 06           PUSH    ES                   also set DS
0000:7C5C 1F           POP     DS                      to zero

                               Alter some of the diskette param table data.

0000:7C5D C645FE0F     MOV     BYTE PTR [DI-02],0F  change head settle time
                                                       at 0000:7c47
0000:7C61 8B0E187C     MOV     CX,[7C18]            sectors per track
0000:7C65 884DF9       MOV     [DI-07],CL              save at 0000:7c42

                               Change INT 1E so that it points to the
                               altered Diskette param table at 0000:7c3e.
```

```
0000:7C68 894702       MOV    [BX+02],AX           change INT 1E segment
0000:7C6B C7073E7C     MOV    WORD PTR [BX],7C3E   change INT 1E offset

                              Call INT 13 with AX=0000, disk reset, so
                              that the new diskette param table is used.

0000:7C6F FB           STI                         interrupts on
0000:7C70 CD13         INT    13                   do diskette reset call
0000:7C72 7279         JB     TALK                 jmp if any error

                              Detemine the starting sector address of
                              the root directory as an LBA.

0000:7C74 33C0         XOR    AX,AX                AX is now zero
0000:7C76 3906137C     CMP    [7C13],AX            number sectros zero?
0000:7C7A 7408         JZ     SMALL_DISK           yes
0000:7C7C 8B0E137C     MOV    CX,[7C13]            number of sectors
0000:7C80 890E207C     MOV    [7C20],CX            save in huge num sects

         SMALL_DISK:

0000:7C84 A0107C       MOV    AL,[7C10]            number of FAT tables
0000:7C87 F726167C     MUL    WORD PTR [7C16]      number of fat sectors
0000:7C8B 03061C7C     ADD    AX,[7C1C]            number of hidden sectors
0000:7C8F 13161E7C     ADC    DX,[7C1E]            number of hidden sectors
0000:7C93 03060E7C     ADD    AX,[7C0E]            number of reserved sectors
0000:7C97 83D200       ADC    DX,+00               number of reserved sectors
0000:7C9A A3507C       MOV    [7C50],AX            save start addr
0000:7C9D 8916527C     MOV    [7C52],DX               of root dir (as LBA)
0000:7CA1 A3497C       MOV    [7C49],AX            save start addr
0000:7CA4 89164B7C     MOV    [7C4B],DX               of root dir (as LBA)

                              Determine sector address of first sector
                              in the data area as an LBA.

0000:7CA8 B82000       MOV    AX,0020              size of a dir entry (32)
0000:7CAB F726117C     MUL    WORD PTR [7C11]      number of root dir entries
0000:7CAF 8B1E0B7C     MOV    BX,[7C0B]            bytes per sector
0000:7CB3 03C3         ADD    AX,BX
0000:7CB5 48           DEC    AX
0000:7CB6 F7F3         DIV    BX
0000:7CB8 0106497C     ADD    [7C49],AX            add to start addr
0000:7CBC 83164B7C00   ADC    WORD PTR [7C4B],+00     of root dir (as LBA)

                              Read the first root dir sector into 0000:0500.

0000:7CC1 BB0005       MOV    BX,0500              addr to read into
0000:7CC4 8B16527C     MOV    DX,[7C52]            get start of address
0000:7CC8 A1507C       MOV    AX,[7C50]               of root dir (as LBA)
0000:7CCB E89200       CALL   CONVERT              call conversion routine
0000:7CCE 721D         JB     TALK                 jmp is any error
0000:7CD0 B001         MOV    AL,01                read 1 sector
0000:7CD2 E8AC00       CALL   READ_SECTORS         read 1st root dir sector
0000:7CD5 7216         JB     TALK                 jmp if any error
0000:7CD7 8BFB         MOV    DI,BX                addr of 1st dir entry
0000:7CD9 B90B00       MOV    CX,000B              count is 11
0000:7CDC BEE67D       MOV    SI,7DE6              addr of file names
0000:7CDF F3           REPZ                        is this "IO.SYS"?
0000:7CE0 A6           CMPSB
0000:7CE1 750A         JNZ    TALK                 no
0000:7CE3 8D7F20       LEA    DI,[BX+20]           addr of next dir entry
0000:7CE6 B90B00       MOV    CX,000B              count is 11
0000:7CE9 F3           REPZ                        is this "MSDOS.SYS"?
0000:7CEA A6           CMPSB
```

```
0000:7CEB 7418         JZ      FOUND_FILES          they are equal

               TALK:

                               Display "Non-System disk..." message,
                               wait for user to hit a key, restore
                               the INT 1E vector and then
                               call INT 19 to start boot processing
                               all over again.

0000:7CED BE9E7D       MOV     SI,7D9E              "Non-System disk..."
0000:7CF0 E85F00       CALL    MSG_LOOP             display message
0000:7CF3 33C0         XOR     AX,AX                INT 16 function
0000:7CF5 CD16         INT     16                   read keyboard
0000:7CF7 5E           POP     SI                   get INT 1E vector's
0000:7CF8 1F           POP     DS                      address
0000:7CF9 8F04         POP     [SI]                 restore the INT 1E
0000:7CFB 8F4402       POP     [SI+02]                 vector's data
0000:7CFE CD19         INT     19                   CALL INT 19 to try again

         SETUP_TALK:

0000:7D00 58           POP     AX                   pop junk off stack
0000:7D01 58           POP     AX                   pop junk off stack
0000:7D02 58           POP     AX                   pop junk off stack
0000:7D03 EBE8         JMP     TALK                 now talk to the user

        FOUND_FILES:

                               Compute the sector address of the first
                               sector of IO.SYS.

0000:7D05 8B471A       MOV     AX,[BX+1A]           get starting cluster num
0000:7D08 48           DEC     AX                   subtract 1
0000:7D09 48           DEC     AX                   subtract 1
0000:7D0A 8A1E0D7C     MOV     BL,[7C0D]            sectors per cluster
0000:7D0E 32FF         XOR     BH,BH
0000:7D10 F7E3         MUL     BX                   multiply
0000:7D12 0306497C     ADD     AX,[7C49]            add start addr of
0000:7D16 13164B7C     ADC     DX,[7C4B]               root dir (as LBA)

                               Read IO.SYS into memory at 0000:0700.  IO.SYS
                               is 3 sectors long.

0000:7D1A BB0007       MOV     BX,0700              address to read into
0000:7D1D B90300       MOV     CX,0003              read 3 sectors

          READ_LOOP:

                               Read the first 3 sectors of IO.SYS
                               (IO.SYS is much longer than 3 sectors).

0000:7D20 50           PUSH    AX                   save AX
0000:7D21 52           PUSH    DX                   save DX
0000:7D22 51           PUSH    CX                   save CX
0000:7D23 E83A00       CALL    CONVERT              call conversion routine
0000:7D26 72D8         JB      SETUP_TALK           jmp if error
0000:7D28 B001         MOV     AL,01                read one sector
0000:7D2A E85400       CALL    READ_SECTORS         read one sector
0000:7D2D 59           POP     CX                   restore CX
0000:7D2E 5A           POP     DX                   restore DX
0000:7D2F 58           POP     AX                   restore AX
0000:7D30 72BB         JB      TALK                 jmp if any INT 13 error
0000:7D32 050100       ADD     AX,0001              add one to the sector addr
```

```
0000:7D35 83D200      ADC     DX,+00               add one to the sector addr
0000:7D38 031E0B7C    ADD     BX,[7C0B]            incr mem addr by sect size
0000:7D3C E2E2        LOOP    READ_LOOP            read next sector

                              Leave information in the AX, BX, CX and DX
                              registers for IO.SYS to use.  Finally,
                              jump to IO.SYS at 0070:0000.

0000:7D3E 8A2E157C    MOV     CH,[7C15]            media type
0000:7D42 8A16247C    MOV     DL,[7C24]            drive number
0000:7D46 8B1E497C    MOV     BX,[7C49]            get start addr of
0000:7D4A A14B7C      MOV     AX,[7C4B]               root dir (as LBA)
0000:7D4D EA00007000  JMP     0070:0000            JUMP TO 0070:0000

               MSG_LOOP:

                              This routine displays a message using
                              INT 10 one character at a time.
                              The message address is in DS:SI.

0000:7D52 AC          LODSB                        get message character
0000:7D53 0AC0        OR      AL,AL                end of message?
0000:7D55 7429        JZ      RETURN               jmp if yes
0000:7D57 B40E        MOV     AH,0E                display one character
0000:7D59 BB0700      MOV     BX,0007              video attrbiutes
0000:7D5C CD10        INT     10                   display one character
0000:7D5E EBF2        JMP     MSG_LOOP             do again

             CONVERT:
                              This routine
                              converts a sector address (an LBA) to
                              a CHS address.  The LBA is in DX:AX.

0000:7D60 3B16187C    CMP     DX,[7C18]            hi part of LBA > sectPerTrk?
0000:7D64 7319        JNB     SET_CARRY            jmp if yes
0000:7D66 F736187C    DIV     WORD PTR [7C18]      div by sectors per track
0000:7D6A FEC2        INC     DL                   add 1 to sector number
0000:7D6C 88164F7C    MOV     [7C4F],DL            save sector number
0000:7D70 33D2        XOR     DX,DX                zero DX
0000:7D72 F7361A7C    DIV     WORD PTR [7C1A]      div number of heads
0000:7D76 8816257C    MOV     [7C25],DL            save head number
0000:7D7A A34D7C      MOV     [7C4D],AX            save cylinder number
0000:7D7D F8          CLC                          clear carry
0000:7D7E C3          RET                          return

           SET_CARRY:

0000:7D7F F9          STC                          set carry

              RETURN:

0000:7D80 C3          RET                          return

        READ_SECTORS:

                              The caller of this routine supplies:
                                 AL = number of sectors to read
                                 ES:BX = memory location to read into
                                 and CHS address to read from in
                                 memory locations 7c25 and 7C4d-7c4f.

0000:7D81 B402        MOV     AH,02                INT 13 read sectors
0000:7D83 8B164D7C    MOV     DX,[7C4D]            get cylinder number
0000:7D87 B106        MOV     CL,06                shift count
```

```
0000:7D89 D2E6        SHL     DH,CL                shift upper cyl left 6 bits
0000:7D8B 0A364F7C    OR      DH,[7C4F]            or in sector number
0000:7D8F 8BCA        MOV     CX,DX                move to CX
0000:7D91 86E9        XCHG    CH,CL                CH=cyl lo, CL=cyl hi + sect
0000:7D93 8A16247C    MOV     DL,[7C24]            drive number
0000:7D97 8A36257C    MOV     DH,[7C25]            head number
0000:7D9B CD13        INT     13                   read sectors
0000:7D9D C3          RET                          return
```

Data not used.

```
0000:7D90 ca86e98a 16247c8a 36257ccd 13c3.... *.....$|.6%|...  *
```

Messages here.

```
0000:7D90 ........ ........ ........ ....0d0a *              ..*
0000:7Da0 4e6f6e2d 53797374 656d2064 69736b20 *Non-System disk *
0000:7Db0 6f722064 69736b20 6572726f 720d0a52 *or disk error..R*
0000:7Dc0 65706c61 63652061 6e642070 72657373 *eplace and press*
0000:7Dd0 20616e79 206b6579 20776865 6e207265 * any key when re*
0000:7De0 6164790d 0a00.... ........ ........ *ady...          *
```

MS DOS hidden file names (first two root directory entries).

```
0000:7De0 ........ ....494f 20202020 20205359 *      IO      SY*
0000:7Df0 534d5344 4f532020 20535953 000055aa *SMSDOS   SYS..U.*
```

The last two bytes contain a 55AAH signature.

```
0000:7Df0 ........ ........ ........ ....55aa *              U.*
```

How It Works -- OS2 Boot Sector

Version 1a

by Hale Landis (landis@sugs.tware.com)

THE "HOW IT WORKS" SERIES

This is one of several How It Works documents. The series currently includes the following:

* How It Works -- CHS Translation
* How It Works -- Master Boot Record
* How It Works -- DOS Floppy Boot Sector
* How It Works -- OS2 Boot Sector
* How It Works -- Partition Tables

OS2 BOOT SECTOR

Note: I'll leave it to someone else to provide you with a disassembly of an OS/2 HPFS boot sector, or a Linux boot sector, or a WinNT boot sector, etc.

This article is a disassembly of a floppy or hard disk boot sector for OS/2. Apparently OS/2 uses the same boot sector for both environments. Basically a bootable FAT hard disk partition looks like a big floppy during the early stages of the system's boot processing. This sector is at cylinder 0, head 0, sector 1 of a floppy or it is the first sector within a FAT hard disk partition. OS/2 floppy disk and hard disk boot sectors are created by the OS/2 FORMAT program.

At the completion of your system's Power On Self Test (POST), INT 19 is called. Usually INT 19 tries to read a boot sector from the first floppy drive. If a boot sector is found on the floppy disk, the that boot sector is read into memory at location 0000:7C00 and INT 19 jumps to memory location 0000:7C00. However, if no boot sector is found on the first floppy drive, INT 19 tries to read the MBR from the first hard drive. If an MBR is found it is read into memory at location 0000:7c00 and INT 19 jumps to memory location 0000:7c00. The small program in the MBR will attempt to locate an active (bootable) partition in its partition table. If such a partition is found, the boot sector of that partition is read into memory at location 0000:7C00 and the MBR program jumps to memory location 0000:7C00. Each operating system has its own boot sector format. The small program in the boot sector must locate the first part of the operating system's kernel loader program (or perhaps the kernel itself or perhaps a "boot manager program") and read that into memory.

INT 19 is also called when the CTRL-ALT-DEL keys are used. On most systems, CTRL-ALT-DEL causes an short version of the POST to be executed before INT 19 is called.

=====

Where stuff is:

```
   The BIOS Parameter Block (BPB) starts at offset 0.
   The boot sector program starts at offset 46.
   The messages issued by this program start at offset 198.
   The OS/2 boot loader file name starts at offset 1d5.
   The boot sector signature is at offset 1fe.
```

Here is a summary of what this thing does:

```
 1) If booting from a hard disk partition, skip to step 6.
 2) Copy Diskette Parameter Table which is pointed to by INT 1E
    to the top of memory.
 3) Alter the copy of the Diskette Parameter Table.
 4) Alter INT 1E to point to altered Diskette Parameter Table at
    the top of memory.
 5) Do INT 13 AH=00, disk reset call so that the altered
    Diskette Parameter Table is used.
 6) Compute sector address of the root directory.
 7) Read the entire root directory into memory starting at
    location 1000:0000.
 8) Search the root directory entires for the file OS2BOOT.
 9) Read the OS2BOOT file into memory at 0800:0000.
10) Do a far return to enter the OS2BOOT program at 0800:0000.
```

NOTES:

This program uses the CHS based INT 13H AH=02 to read the FAT root directory and to read the OS2BOOT file. If the drive is >528MB, this CHS must be a translated CHS (or L-CHS, see my BIOS TYPES document). Except for internal computations no addresses in LBA form are used, another reason why LBA doesn't solve the >528MB problem.

=====

Here is the entire sector in hex and ascii.

```
OFFSET 0 1 2 3  4 5 6 7  8 9 A B  C D E F  *0123456789ABCDEF*
000000 eb449049 424d2032 302e3000 02100100 *.D.IBM 20.0.....*
000010 02000200 00f8d800 3e000e00 3e000000 *........>...>...*
000020 06780d00 80002900 1c0c234e 4f204e41 *.x....)...#NO NA*
000030 4d452020 20204641 54202020 20200000 *ME    FAT     ..*
000040 00100000 0000fa33 db8ed3bc ff7bfbba *.......3.....{..*
000050 c0078eda 803e2400 00753d1e b840008e *.....>$..u=..@..*
000060 c026ff0e 1300cd12 c1e0068e c033ff33 *.&...........3.3*
000070 c08ed8c5 367800fc b90b00f3 a41fa118 *....6x..........*
000080 0026a204 001e33c0 8ed8a378 008c067a *.&....3....x...z*
000090 001f8a16 2400cd13 a0100098 f7261600 *....$........&..*
0000a0 03060e00 5091b820 00f72611 008b1e0b *....P.. ..&.....*
0000b0 0003c348 f7f35003 c1a33e00 b800108e *...H..P...>.....*
0000c0 c033ff59 890e4400 58a34200 33d2e873 *.3.Y..D.X.B.3..s*
0000d0 0033db8b 0e11008b fb51b90b 00bed501 *.3.......Q......*
0000e0 f3a65974 0583c320 e2ede335 268b471c *..Yt... ...5&.G.*
0000f0 268b571e f7360b00 fec08ac8 268b571a *&.W..6......&.W.*
000100 4a4aa00d 0032e4f7 e203063e 0083d200 *JJ...2.....>....*
000110 bb00088e c333ff06 57e82800 8d360b00 *.....3..W.(..6..*
000120 cbbe9801 eb03bead 01e80900 bec201e8 *................*
000130 0300fbeb feac0ac0 7409b40e bb0700cd *........t.......*
000140 10ebf2c3 50525103 061c0013 161e00f7 *....PRQ.........*
000150 361800fe c28ada33 d2f7361a 008afa8b *6......3..6.....*
000160 d0a11800 2ac34050 b402b106 d2e60af3 *....*.@P........*
000170 8bca86e9 8a162400 8af78bdf cd1372a6 *......$.......r.*
000180 5b598bc3 f7260b00 03f85a58 03c383d2 *[Y...&....ZX....*
000190 002acb7f afc31200 4f532f32 20212120 *.*......OS/2 !! *
0001a0 53595330 31343735 0d0a0012 004f532f *SYS01475.....OS/*
0001b0 32202121 20535953 30323032 350d0a00 *2 !! SYS02025...*
0001c0 12004f53 2f322021 21205359 53303230 *..OS/2 !! SYS020*
0001d0 32370d0a 004f5332 424f4f54 20202020 *27...OS2BOOT    *
0001e0 00000000 00000000 00000000 00000000 *................*
0001f0 00000000 00000000 00000000 000055aa *..............U.*
```

=====

The first 62 bytes of a boot sector are known as the BIOS Parameter Block (BPB). Here is the layout of the BPB fields and the values they are assigned in this boot sector:

```
   db JMP instruction      at 7c00 size  2 = eb44
   db NOP instruction         7c02       1   90
   db OEMname                 7c03       8   'IBM 20.0'
   dw bytesPerSector          7c0b       2   0200
   db sectPerCluster          7c0d       1   01
   dw reservedSectors         7c0e       2   0001
   db numFAT                  7c10       1   02
   dw numRootDirEntries       7c11       2   0200
   dw numSectors              7c13       2   0000 (use numSectorsHuge)
   db mediaType               7c15       1   f8
   dw numFATsectors           7c16       2   00d8
   dw sectorsPerTrack         7c18       2   003e
   dw numHeads                7c1a       2   000e
   dd numHiddenSectors        7c1c       4   00000000
   dd numSectorsHuge          7c20       4   000d7806
   db driveNum                7c24       1   80
   db reserved                7c25       1   00
   db signature               7c26       1   29
   dd volumeID                7c27       4   001c0c23
   db volumeLabel             7c2b      11   'NO NAME    '
   db fileSysType             7c36       8   'FAT     '
```

```
=====

Here is the boot sector...

The first 3 bytes of the BPB are JMP and NOP instructions.

0000:7C00 EB44        JMP     START
0000:7C02 90          NOP

Here is the rest of the BPB.

0000:7C00 eb449049 424d2032 302e3000 02100100 *.D.IBM 20.0.....*
0000:7C10 02000200 00f8d800 3e000e00 3e000000 *........>...>...*
0000:7C20 06780d00 80002900 1c0c234e 4f204e41 *.x....)...#NO NA*
0000:7C30 4d452020 20204641 54202020 20200000 *ME    FAT     ..*

Additional data areas.

0000:7C30 ........ ........ ........ ....0000 *              ..*
0000:7C40 00100000 0000.... ........ ........ *......          *

   Note:

   0000:7c3e (DS:003e) = number of sectors in the FATs and root dir.
   0000:7c42 (DS:0042) = number of sectors in the FAT.
   0000:7c44 (DS:0044) = number of sectors in the root dir.

               START:                         START OF BOOT SECTOR PROGRAM

0000:7C46 FA          CLI                           interrupts off
0000:7C47 33DB        XOR     BX,BX                 zero BX
0000:7C49 8ED3        MOV     SS,BX                 SS now zero
0000:7C4B BCFF7B      MOV     SP,7BFF               SP now 7bff
0000:7C4E FB          STI                           interrupts on
0000:7C4F BAC007      MOV     DX,07C0               set DX to
0000:7C52 8EDA        MOV     DS,DX                    07c0

                              Are we booting from a floppy or a
                              hard disk partition?

0000:7C54 803E240000  CMP     BYTE PTR [0024],00    is driveNum in BPB 00?
0000:7C59 753D        JNZ     NOT_FLOPPY            jmp if not zero

                              We are booting from a floppy.  The
                              Diskette Parameter Table must be
                              copied and altered...

   Diskette Parameter Table is pointed to by INT 1E.  This
   program moves this table to high memory, alters the table, and
   changes INT 1E to point to the altered table.

   This table contains the following data:

   ????:0000 = Step rate and head unload time.
   ????:0001 = Head load time and DMA mode flag.
   ????:0002 = Delay for motor turn off.
   ????:0003 = Bytes per sector.
   ????:0004 = Sectors per track.
   ????:0005 = Intersector gap length.
   ????:0006 = Data length.
   ????:0007 = Intersector gap length during format.
   ????:0008 = Format byte value.
   ????:0009 = Head settling time.
   ????:000a = Delay until motor at normal speed.
```

```
                                   Compute a valid high memory address.

0000:7C5B 1E           PUSH    DS                    save DS
0000:7C5C B84000       MOV     AX,0040               set ES
0000:7C5F 8EC0         MOV     ES,AX                    to 0040 (BIOS data area)
0000:7C61 26           ES:                           reduce system memory
0000:7C62 FF0E1300     DEC     WORD PTR [0013]          size by 1024
0000:7C66 CD12         INT     12                    get system memory size
0000:7C68 C1E06        SHL     AX,06                 shift AX (mult by 64)
0000:7C6B 8EC0         MOV     ES,AX                 move to ES
0000:7C6D 33FF         XOR     DI,DI                 zero DI

                               Move the diskette param table to high memory.

0000:7C6F 33C0         XOR     AX,AX                 zero AX
0000:7C71 8ED8         MOV     DS,AX                 DS now zero
0000:7C73 C5367800     LDS     SI,[0078]             DS:SI = INT 1E vector
0000:7C77 FC           CLD                           clear direction
0000:7C78 B90B00       MOV     CX,000B               count is 11
0000:7C7B F3           REPZ                          copy diskette param table
0000:7C7C A4           MOVSB                            to top of memory

                               Alter the number of sectors per track
                               in the diskette param table in high memory.

0000:7C7D 1F           POP     DS                    restore DS
0000:7C7E A11800       MOV     AX,[0018]             get sectorsPerTrack from BPB
0000:7C81 26           ES:                           alter sectors per track
0000:7C82 A20400       MOV     [0004],AL                in diskette param table

                               Change INT 1E to point to altered diskette
                               param table and do a INT 13 disk reset call.

0000:7C85 1E           PUSH    DS                    save DS
0000:7C86 33C0         XOR     AX,AX                 AX now zero
0000:7C88 8ED8         MOV     DS,AX                 DS no zero
0000:7C8A A37800       MOV     [0078],AX             alter INT 1E vector
0000:7C8D 8C067A00     MOV     [007A],ES                to point to altered
                                                        diskette param table
0000:7C91 1F           POP     DS                    restore DS
0000:7C92 8A162400     MOV     DL,[0024]             driveNum from BPB
0000:7C96 CD13         INT     13                    diskette reset

         NOT_FLOPPY:

                               Compute the location and the size of
                               the root directory.  Read the entire
                               root directory into memory.

0000:7C98 A01000       MOV     AL,[0010]             get numFAT
0000:7C9B 98           CBW                           make into a word
0000:7C9C F7261600     MUL     WORD PTR [0016]       mult by numFatSectors
0000:7CA0 03060E00     ADD     AX,[000E]             add reservedSectors
0000:7CA4 50           PUSH    AX                    save
0000:7CA5 91           XCHG    CX,AX                 move to CX
0000:7CA6 B82000       MOV     AX,0020               dir entry size
0000:7CA9 F7261100     MUL     WORD PTR [0011]       mult by numRootDirEntries
0000:7CAD 8B1E0B00     MOV     BX,[000B]             get bytesPerSector
0000:7CB1 03C3         ADD     AX,BX                 add
0000:7CB3 48           DEC     AX                    subtract 1
0000:7CB4 F7F3         DIV     BX                    div by bytesPerSector
0000:7CB6 50           PUSH    AX                    save number of dir sectors
0000:7CB7 03C1         ADD     AX,CX                 add number of fat sectors
```

```
0000:7CB9 A33E00       MOV    [003E],AX            save
0000:7CBC B80010       MOV    AX,1000              AX is now 1000
0000:7CBF 8EC0         MOV    ES,AX                ES is now 1000
0000:7CC1 33FF         XOR    DI,DI                DI is now zero
0000:7CC3 59           POP    CX                   get number dir sectors
0000:7CC4 890E4400     MOV    [0044],CX            save
0000:7CC8 58           POP    AX                   get number fat sectors
0000:7CC9 A34200       MOV    [0042],AX            save
0000:7CCC 33D2         XOR    DX,DX                DX now zero
0000:7CCE E87300       CALL   READ_SECTOR          read 1st sect of root dir
0000:7CD1 33DB         XOR    BX,BX                BX is now zero
0000:7CD3 8B0E1100     MOV    CX,[0011]            number of root dir entries

         DIR_SEARCH:                          SEARCH FOR OS2BOOT.

                              Search the root directory for the file
                              name OS2BOOT.

0000:7CD7 8BFB         MOV    DI,BX                DI is dir entry addr
0000:7CD9 51           PUSH   CX                   save CX
0000:7CDA B90B00       MOV    CX,000B              count is 11
0000:7CDD BED501       MOV    SI,01D5              addr of "OS2BOOT"
0000:7CE0 F3           REPZ                        is 1st dir entry
0000:7CE1 A6           CMPSB                          for "OS2BOOT"?
0000:7CE2 59           POP    CX                   restore CX
0000:7CE3 7405         JZ     FOUND_OS2BOOT        jmp if OS2BOOT
0000:7CE5 83C320       ADD    BX,+20               incr to next dir entry
0000:7CE8 E2ED         LOOP   DIR_SEARCH           try again

      FOUND_OS2BOOT:                          FOUND OS2BOOT.

                              OS2BOOT was found.  Get the starting
                              cluster number and convert to a sector
                              address.  Read OS2BOOT into memory and
                              finally do a far return to enter
                              the OS2BOOT program.

0000:7CEA E335         JCXZ   FAILED1              JMP if CX zero.
0000:7CEC 26           ES:                         get the szie of
0000:7CED 8B471C       MOV    AX,[BX+1C]              the OS2BOOT file
0000:7CF0 26           ES:                            from the OS2BOOT
0000:7CF1 8B571E       MOV    DX,[BX+1E]              directory entry
0000:7CF4 F7360B00     DIV    WORD PTR [000B]      div by bytesPerSect
0000:7CF8 FEC0         INC    AL                   add 1
0000:7CFA 8AC8         MOV    CL,AL                num sectors OS2BOOT
0000:7CFC 26           ES:                         get the starting
0000:7CFD 8B571A       MOV    DX,[BX+1A]              cluster number
0000:7D00 4A           DEC    DX                   subtract 1
0000:7D01 4A           DEC    DX                   subtract 1
0000:7D02 A00D00       MOV    AL,[000D]            sectorsPerCluster
0000:7D05 32E4         XOR    AH,AH                mutiply
0000:7D07 F7E2         MUL    DX                      to get LBA
0000:7D09 03063E00     ADD    AX,[003E]            add number of FAT sectors
0000:7D0D 83D200       ADC    DX,+00                  to LBA
0000:7D10 BB0008       MOV    BX,0800              set ES
0000:7D13 8EC3         MOV    ES,BX                   to 0800
0000:7D15 33FF         XOR    DI,DI                set ES:DI to entry point
0000:7D17 06           PUSH   ES                      address of
0000:7D18 57           PUSH   DI                         OS2BOOT
0000:7D19 E82800       CALL   READ_SECTOR          read OS2BOOT into memory
0000:7D1C 8D360B00     LEA    SI,[000B]            set DS:SI
0000:7D20 CB           RETF                        "far return" to OS2BOOT

            FAILED1:                              OS2BOOT WAS NOT FOUND.
```

```
0000:7D21 BE9801      MOV     SI,0198              "SYS01475" message
0000:7D24 EB03        JMP     FAILED3

            FAILED2:                            ERROR FROM INT 13.

0000:7D26 BEAD01      MOV     SI,01AD              "SYS02025" message

            FAILED3:                            OUTPUT ERROR MESSAGES.

0000:7D29 E80900      CALL    MSG_LOOP             display message
0000:7D2C BEC201      MOV     SI,01C2              "SYS02027" message
0000:7D2F E80300      CALL    MSG_LOOP             display message
0000:7D32 FB          STI                          interrupts on

               HANG:                            HANG THE SYSTEM!

0000:7D33 EBFE        JMP     HANG                 sit and stay!

           MSG_LOOP:                         DISPLAY AN ERROR MESSAGE.

                              Routine to display the message
                              text pointed to by SI.

0000:7D35 AC          LODSB                        get next char of message
0000:7D36 0AC0        OR      AL,AL                end of message?
0000:7D38 7409        JZ      RETURN               jmp if yes
0000:7D3A B40E        MOV     AH,0E                write 1 char
0000:7D3C BB0700      MOV     BX,0007              video attributes
0000:7D3F CD10        INT     10                   INT 10 to write 1 char
0000:7D41 EBF2        JMP     MSG_LOOP             do again

             RETURN:

0000:7D43 C3          RET                          return

       READ_SECTOR:                          ROUTINE TO READ SECTORS.

                              Read sectors into memory.  Read multiple
                              sectors but don't read across a track
                              boundary.

                              The caller supplies the following:
                                 DX:AX = sector address to read (as LBA)
                                    CX = number of sectors to read
                                 ES:DI = memory address to read into

0000:7D44 50          PUSH    AX                   save lower part of LBA
0000:7D45 52          PUSH    DX                   save upper part of LBA
0000:7D46 51          PUSH    CX                   save number of sect to read
0000:7D47 03061C00    ADD     AX,[001C]            add numHiddenSectors
0000:7D4B 13161E00    ADC     DX,[001E]               to LBA
0000:7D4F F7361800    DIV     WORD PTR [0018]      div by sectorsPerTrack
0000:7D53 FEC2        INC     DL                   add 1 to sector number
0000:7D55 8ADA        MOV     BL,DL                save sector number
0000:7D57 33D2        XOR     DX,DX                zero upper part of LBA
0000:7D59 F7361A00    DIV     WORD PTR [001A]      div by numHeads
0000:7D5D 8AFA        MOV     BH,DL                save head number
0000:7D5F 8BD0        MOV     DX,AX                save cylinder number
0000:7D61 A11800      MOV     AX,[0018]            sectorsPerTrack
0000:7D64 2AC3        SUB     AL,BL                sub sector number
0000:7D66 40          INC     AX                   add 1
0000:7D67 50          PUSH    AX                   save number of sector to rea
d
```

```
0000:7D68 B402        MOV    AH,02               INT 13 read sectors
0000:7D6A B106        MOV    CL,06               shift count
0000:7D6C D2E6        SHL    DH,CL               shift high cyl left
0000:7D6E 0AF3        OR     DH,BL               or in sector number
0000:7D70 8BCA        MOV    CX,DX               move cyl/sect to CX
0000:7D72 86E9        XCHG   CH,CL               swap cyl/sect
0000:7D74 8A162400    MOV    DL,[0024]           driveNum
0000:7D78 8AF7        MOV    DH,BH               head number
0000:7D7A 8BDF        MOV    BX,DI               memory addr to read into
0000:7D7C CD13        INT    13                  INT 13 read sectors call
0000:7D7E 72A6        JB     FAILED2             jmp if any error
0000:7D80 5B          POP    BX                  get number of sectors read
0000:7D81 59          POP    CX                  restore CX
0000:7D82 8BC3        MOV    AX,BX               number of sector to AX
0000:7D84 F7260B00    MUL    WORD PTR [000B]     multiply by sector size
0000:7D88 03F8        ADD    DI,AX               add to memory address
0000:7D8A 5A          POP    DX                  restore upper part of LBA
0000:7D8B 58          POP    AX                  resotre lower part of LBA
0000:7D8C 03C3        ADD    AX,BX               add number of sector just
0000:7D8E 83D200      ADC    DX,+00                 read to LBA
0000:7D91 2ACB        SUB    CL,BL               decr requested num of sect
0000:7D93 7FAF        JG     READ_SECTOR         jmp if not zero
0000:7D95 C3          RET                        return
```

Data not used.

```
0000:7D90 ........ ....1200 ........ ........ *       ..        *
```

Messages here.

```
0000:7D90 ........ ........ 4f532f32 20212120 *         OS/2 !! *
0000:7Da0 53595330 31343735 0d0a0012 004f532f *SYS01475.....OS/*
0000:7Db0 32202121 20535953 30323032 350d0a00 *2 !! SYS02025...*
0000:7Dc0 12004f53 2f322021 21205359 53303230 *..OS/2 !! SYS020*
0000:7Dd0 32370d0a 00...... ........ ........ *27...           *
```

OS/2 loader file name.

```
0000:7Dd0 ........ ..4f5332 424f4f54 20202020 *     OS2BOOT    *
```

Data not used.

```
0000:7De0 00000000 00000000 00000000 00000000 *................*
0000:7Df0 00000000 00000000 00000000 0000.... *..............  *
```

The last two bytes contain a 55AAH signature.

```
0000:7Df0 ........ ........ ........ ....55aa *              U.*
```