Disk access using the BIOS (INT 13h)

The BIOS provides a set of disk access routines using the INT 0x13 family of BIOS functions. Until an appropriate driver is implemented, these BIOS functions are the only way to access disks.

There are two basic INT 0x13 call families to use for disk access. One uses Cylinder, Head, Sector addressing, and the other uses LBA addressing. There is a third command set to access ATAPI drives using the PACKET command set.

Note: These BIOS INT calls should completely preserve all the registers (except AX). However, some older versions of the Bochs/QEMU BIOS destroyed the upper 16-bit values of some of the 32bit registers.

CHS

Remember that there are severe addressing limitations with CHS addressing. Typically only the first 8 GB of the media can be accessed, at most. Sometimes USB drives might be using floppy emulation, so the maximum accessable size becomes only 1.44M.

Converting LBA to CHS

The addresses of sectors on a disk are almost always calculated or stored as LBAs, but some drives (especially floppies and USB drives with floppy emulation) cannot use LBA addressing. So your code must translate the address, and use a CHS read call. This also applies if you are trying to read floppies and hard disks with the same code.

Quick Explanation of the Algorithm

You can think of a CHS address as the digits of a 3-digit number. The sectors are the low digit, the heads are the middle digit, and cylinders are the high digit. As an analogy, think of the decimal number 345. To extract the low (sectors) digit, you take the modulo with 10. 345 % 10 = 5. You also need the integer result of 345 / 10 to calculate the heads and cylinders. 345 / 10 = 34. Then %10 again gets the head, and /10 gets the cylinder. The nice thing is that all CPU chips have "div" opcodes that give you **both** the result **and** the modulus for every operation.

The Algorithm

LBA is the input value,

- Temp = LBA / (Sectors per Track)
- Sector = (LBA % (Sectors per Track)) + 1
- Head = Temp % (Number of Heads)
- Cylinder = Temp / (Number of Heads)

Note: Always remember that sector is 1-based, and not 0-based ... this detail causes many problems.

Getting Sectors/Track, Total Head values

There is only one real place where you can get the "Sectors per Track" and "Number of Heads" values for the previous LBA->CHS calculation. All modern BIOSes use automatic CHS to LBA conversions internally, with internal hardcoded conversion values. They **do not use** the "real" CHS values that are written on the drive's label. Also, if you perhaps have a FAT formatted drive, it will claim to have "Sectors per Track" and "Number of Heads" information stored in the BPB. These values are almost always **wrong**.

If the 0x80 bit is set for the BIOS drive number, then you have no real choice other than to use the INT13h AH=8 BIOS function to get the "drive geometry".

- Set AH to 8, DL to the BIOS drive number, and execute INT 0x13.
- The value returned in DH is the "Number of Heads" -1.
- AND the value returned in CL with 0x3f to get the "Sectors per Track".

Note: INT 0x13 AH=8 does not work well with floppy drives, or emulated floppy drives. It may be best to use default values in that case.

Reading sectors with a CHS address

Cylinder = 0 to 1023 (maybe 4095), Head = 0 to 15 (maybe 254, maybe 255), Sector = 1 to 63

- Set AH = 2
- AL = total sector count (0 is illegal) -- cannot cross ES page boundary, **or a cylinder boundary**, and must be < 128
- CH = cylinder & 0xff
- CL = Sector | ((cylinder >> 2) & 0xC0);
- DH = Head -- may include two more cylinder bits
- ES:BX -> buffer
- Set DL = "drive number" -- typically 0x80, for the "C" drive
- Issue an INT 0x13.

The carry flag will be set if there is any error during the read. AH should be set to 0 on success.

To write: set AH to 3, instead.

Note: The limitation about not crossing cylinder boundaries is very annoying, especially when combined with the 127 sector limit -- because the arithmetic for the length and "start CHS" of the *next* consecutive read or write gets messy. The simplest workaround is to read or write only one sector at a time in CHS mode. Not all BIOSes have these two limitations, of course, but it is necessary to program for the "lowest common

denominator".

LBA in Extended Mode

To use LBA addressing with INT 0x13, you need to use a command in the "INT13h Extensions". Every BIOS since the mid-90's supports the extensions, but you may want to verify that they are supported anyway.

- Set AH = 0x41
- BX = 0x55AA
- DL = 0x80
- Issue an INT 0x13.

The carry flag will be set if Extensions are **not** supported.

To read or write, first you need to set up a "Disk Address Packet Structure" in memory, on a uint32_t (4 byte) boundary.

- Format of disk address packet:

Offset	Size	Description
0	1	size of packet (16 bytes)
1	1	always 0
2	2	number of sectors to transfer (max 127 on some BIOSes)
4	4	transfer buffer (16 bit segment:16 bit offset) (see note #1)
8	4	lower 32-bits of 48-bit starting LBA
12	4	upper 16-bits of 48-bit starting LBA

Notes:

(1) The 16 bit segment value ends up at an offset of 6 from the beginning of the structure (i.e., when declaring segment:offset as two separate 16-bit fields, place the offset first and then follow with the segment because x86 is little-endian).

(2) If the disk drive itself does not support LBA addressing, the BIOS will automatically convert the LBA to a CHS address for you -- so this function still works.

(3) The transfer buffer should be 16-bit (2 byte) aligned.

To read a disk:

- Set the proper values in the disk address packet structure
- Set DS:SI -> Disk Address Packet in memory
- Set AH = 0x42
- Set DL = "drive number" -- typically 0x80 for the "C" drive
- Issue an INT 0x13.

The carry flag will be set if there is any error during the transfer. AH should be set to 0 on success.

To write to a disk, set AH = 0x43.

64-bit Addressing Extensions

The BIOS Enhanced Disk Drives Services version 3.0 specifies the possibility of loading to flat 64-bit addresses. Then the format of disk address packet looks like:

Offset	Size	Description
0	1	size of packet (24 bytes)
1	1	always 0
2	2	number of sectors to transfer (max 127 on some BIOSes)
4	4	transfer buffer (0xFFFF:0xFFFF)
8	4	lower 32-bits of starting 48-bit LBA
12	4	upper 32-bits of starting 48-bit LBA
16	4	lower 32-bits of load address
20	4	upper 32-bits of load address

Warning: 64-bit addressing extensions are not implemented by most BIOSes, therefore they are very unpractical. Even BIOSes of most virtual machine softwares don't implement them!

x86 Examples

- Reading 16 sectors from LBA #1 to physical address 0x7C00

```
DAPACK:
        db    0x10
        db    0
blkcnt:    dw    16          ; int 13 resets this to # of blocks actually read/written
db_add:    dw    0x7C00          ; memory buffer destination address (0:7c00)
        dw    0           ; in memory page zero
d_lba:     dd    1           ; put the lba to read in this spot
        dd    0           ; more storage bytes only for big lba's ( > 4 bytes )

        mov si, DAPACK          ; address of "disk address packet"
        mov ah, 0x42        ; AL is unused
        mov dl, 0x80        ; drive number 0 (OR the drive # with 0x80)
        int 0x13
        jc short .error
```

Supported Systems

All systems support CHS addressing.

There exist some 486 systems that do not support LBA in any way. All known original Pentium and newer systems support Extended LBA in the BIOS.

One of the easiest ways to read or write a USB flash drive is to drop into Real or Unreal Mode, and use the INT 0x13 BIOS commands. However, the transfer must fit in the usable part of low memory (if in Real Mode), and you need to somehow know the proper drive number to use in DL.

BIOS can emulate USB flash drives as a floppy drive or as an hard disk. If the USB drive is emulated as a floppy drive, it is likely 0x00 or 0x01 and you can only use AH=0x02 and AH=0x03. If it's emulated as an hard disk, it is most likely 0x80 or 0x81 and you can **also** use AH=0x42 and AH=0x43.