DAS BOOT

First things first: in order for a computer to run xv6, we need to load it from disk into memory and tell the processor to start running it. So how does this all happen?

The Boot Process

When you press the power button, the hardware gets initialized by a piece of firmware called the BIOS (Basic Input/Output System) that comes pre-installed on the motherboard on a ROM chip. Nowadays, your computer probably uses UEFI loaded from flash memory, but xv6 pretends like it's 1995 and sticks with BIOS. Since xv6 runs on x86 hardware, we're gonna have to satisfy all the janky requirements that come with that architecture, in addition to the BIOS's requirements.

Now the BIOS has to load some *other* code called the boot loader from disk; then it's the boot loader's job to load the OS and get it running. The boot loader has to act as a middle-man because the BIOS has no idea where on the disk you decided to put the OS.

The BIOS will look for the boot loader in the very first sector (512 bytes) of whatever mass storage device you told it to boot from, which we'll call the boot disk. The processor will execute the instructions it finds there. This means you have to make a choice: either your boot loader has to be less than 512 bytes or you can split it up into smaller parts and have each part load the next one. xv6 takes the first approach.

The BIOS loads the boot loader into memory at address 0x7C00, then sets the processor's `%ip` register to that address and jumps to it. Remember that `%eip` is the instruction pointer on x86? Okay cool. But why did I write `%ip` instead of `%eip`? Well, the BIOS assumes we're gonna be using 16 bits because of the hellscape known as backwards-compatibility, so we've gotta pretend like it's 1975 before we can pretend it's 1995. The irony here is that this initial 16-bit mode is called "real mode". So on top of loading the OS, the boot loader will also have to shepherd the processor from real mode to 32-bit "protected mode".

One last detail: we'll look at the Makefile and linker script later on, but for now just keep in mind that the boot loader will be compiled separately from the kernel, which will be compiled separately from all the user-space programs. This makes it easier to make sure that the entire boot loader will fit in the first 512 bytes on disk. Eventually, the boot loader and the kernel will be stored on the same boot disk together, and the user-space programs will be on a separate disk that holds the file system.

bootasm.S

Boot loader space is tight, and we want to make sure our instructions are exact, so we're gonna start off in assembly. The ".S" file extension means it's gonna be assembled by the GNU assembler `as`, and we're allowed to use C preprocessor directives like `#include` or `#define` or whatever in the assembly code. Also, xv6 uses AT&T syntax, so if you read CS:APP or took the online course then it'll be familiar; if you don't know what that means, then don't worry about it.

Getting Started

First we include some header files to use some constants; I'll point them out later. Next up, we gotta tell the assembler to generate 16-bit code, and set a global label to tell the BIOS where to start executing code.

```
.code16         # Tell compiler to generate 16-bit code
.globl start
start:
```

Next up: you know how sometimes you can press a special key to tell the BIOS to stop what it's doing and let you pick a disk to boot from? Or you move your mouse around in the BIOS menu and you see the pointer moving? Yeah, that needs hardware interrupts in order to work, but right now, we don't have the faintest clue how to handle those if they happen, so let's go ahead and turn those off. There's an x86 instruction to disable them by clearing the interrupt flag in the CPU's flags register.

```
    cli
```

Now we've gotta handle some of x86's quirks. First off, we're gonna need 20-bit memory addresses, but we only have 16 bits to work with. x86 uses six segment registers `%cs` (code segment), `%ds` (data segment), `%ss` (stack segment), `%es` (extra segment), `%fs` and `%gs` (general-purpose segments) to create 20-bit addresses from 16-bit ones; we're gonna need the first four. The BIOS guarantees that `%cs` will be set to zero, but it doesn't make any promises about the others, so we have to clear them ourselves. We're not using `%eax` for anything yet, so we'll use that to clear the others. The `w` at the end of `xorw` and `movw` means we're operating on 16-bit words.

```
    xorw    %ax,%ax
    movw    %ax,%ds     # Data segment
    movw    %ax,%es     # Extra segment
    movw    %ax,%ss     # Stack segment
```

This next part is a total hack for backwards-compatibility: sometimes a virtual address might get converted to a 21-bit physical address, and oh no, what are we gonna do? Well, some hardware can't deal with 21 bits, so it just ignores it, but it's 1995, so we've got fancy hardware that can use that extra bit. Wow, you really know we're in the future when you've got a whole 2 MB of RAM to work with! So we have to tell the processor not to throw away that 21st bit. The way we do that is by setting the second bit of the keyboard controller's output port to line high. I don't know. Don't ask me why. The output ports are 0x64 and 0x60, so we're gonna wait until they're not busy, then set the magic values that will make this all work.

```
seta20.1:
    inb     $0x64,%al   # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.1

    movb    $0xd1,%al   # 0xD1 -> port 0x64
    outb    %al,$0x64

seta20.2:
    inb     $0x64,%al   # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.2

    movb    $0xdf,%al   # 0xDF -> port 0x60
    outb    %al,$0x60
```

Segmentation

Now it's time to switch to 32-bit "protected mode". Up until now, the processor has been converting virtual addresses to physical ones using those segment registers which we cleared, so the mapping has been an identity map. But let's talk about how x86 converts 32-bit virtual addresses to physical ones; this is important for the rest of the boot loader code as well as the OS, so you're gonna have to bear with me for this maelstrom of x86-specific details.

The x86 architecture does the conversion in two steps: first segmentation, then paging. A virtual address starts off life as a *logical address*. Segmentation converts that to a *linear address*, and paging converts that to a physical one.

A logical address consists of a 20-bit *segment selector* and a 12-bit offset, with the segment bits before the offset bits, like `segment:offset`. The CPU's segmentation hardware uses those segment bits to pick one of those four segment registers we cleared earlier, which acts as an index into a *Global Descriptor Table* or GDT. Each entry of this GDT tells you where that segment is found in memory using a base physical address and a virtual address for the maximum or limit.

The GDT entry also has some permission bits for that segment; the segmentation hardware will check whether each address can be written to and whether the process generating the virtual address has the right permissions to access it. These checks compare the GDT entry's *Descriptor Privilege Levels*, also known as *ring levels*, against the *Current Privilege Level*. x86 has four privilege levels (0-3), so if you've ever heard of the kernel operating in ring 0 or user code in ring 3, this is where it comes from.

Okay, so the GDT entry will give us the first 20 bits of the new linear address; the offset bits stay the same. After that, the linear address is ready to be converted to a physical address by the paging hardware. We'll go over this second half of the story in the virtual memory section. For now, the point is this: xv6 is mostly gonna say no thank you to segmentation and stick to paging alone for memory virtualization.

So we're gonna set up our GDT to map all segments the exact same way: with a base of zero and the maximum possible limit (with 32 bits, that works out to a grand total of 4 GB, wow so much RAM, I can't imagine ever needing more). We have to stick this GDT somewhere in our code so we can point the CPU to it, so we'll put it at the end and throw a `gdtdesc` label on it. Now we can tell the CPU to load it up with a special x86 instruction for that.

```
    lgdt    gdtdesc
```

Protected Mode

Good news, everyone! We're finally ready to turn on protected mode, which we do by setting the zero bit of the `%cr0` control register. Note that the `l` at the end of the instructions here means we're now using long words, i.e. 32 bits; `CR0_PE` is defined in the mmu.h
header file as 0x1.

```
    movl    %cr0, %eax      # Copy %cr0 into %eax
    orl     $CR0_PE, %eax   # Set bit 0
    movl    %ax, %cr0       # Copy it back
```

Oh wait, I lied. Enabling protection mode like we just did doesn't change how the processor translates addresses. We have to load a new value into a segment register to make the CPU read the GDT and change its internal segmentation settings. We can do that by using a long jump instruction, which lets us specify a code segment selector. We're just gonna jump to the very next line anyway, but in doing so we'll force the CPU to start using the GDT, which describes a 32-bit code segment, so *now* we're finally in 32-bit mode! Here, `SEG_KCODE` is a constant defined in mmu.h as segment 1, for `%cs`; we bitshift it left by 3.

```
    ljmp    $(SEG_KCODE<<3), $start32
```

First we signal the compiler to start generating 32-bit code. Then we initialize the data, extra, and stack segment registers to point to the `SEG_KDATA` entry of the GDT; that constant is defined in mmu.h as the segment for the kernel data and stack. We're not required to set up `%fs` and `%gs`, so we'll just zero them.

```
.code 32    # Tell assembler to generate 32-bit code now
start32:
    movw    $(SEG_KDATA<<3), %ax    # Our data segment selector
    movw    %ax, %ds    # Data segment
    movw    %ax, %es    # Extra segment
    movw    %ax, %ss    # Stack segment
    movw    $0, %ax     # Zero the segments not ready for use
    movw    %ax, %fs
    movw    %ax, %gs
```

The Kernel Stack

Okay, last step in the assembly code now: we have to set up a stack in an unused part of memory. In x86, the stack grows downwards, so the "top" of the stack--that is, the most-recently-added byte--is actually at the bottom of the stack in physical memory. It's annoying, but we're gonna have to keep track of that. The `%ebp` register points to the base of the stack (i.e., the first byte we pushed onto the stack), and the `%esp` register holds the address of the top of the stack (most-recently-pushed byte).

But where should we put the stack? The memory from 0xA_0000 to 0x10_0000 is littered with a memory regions that I/O devices are gonna be checking, so that's out. The boot loader starts at 0x7C00 and takes up 512 bytes, so that means it ends at 0x7E00. So xv6 is gonna start the stack at 0x7C00 and have it grow down from there, toward 0x0000 and away from the boot loader. Remember how back in the beginning, we started off the assembly code with a `start` label? That means that `start` is conveniently located at 0x7C00.

```
    movl    $start, %esp
```

And we're done with assembly! Time to move on to C code for the rest of the boot loader. We'll take over with a C function called `bootmain()`, which should never return. The linker will take care of connecting the call here to its definition in bootmain.c.

```
    call    bootmain
```

Handling Errors

Wait, what? There's more assembly code after this? Why?

Well, if something goes wrong in `bootmain()`, then the function will return, so we have to handle that here. Since we usually run OSes we're developing in an emulator like Bochs or QEMU, we'll trigger a breakpoint and loop. Bochs listens on port 0x8A00, so we can transfer control back to it there; this wouldn't do anything on real hardware.

```
    movw    $0x8a00, %ax    # 0x8a00 -> port 0x8a00
    movw    %ax, %dx
    outw    %ax, %dx
    movw    $0x8ae0, %ax    # 0x8ae0 -> port 0x8a00
    outw    %ax, %dx
spin:
    jmp     spin            # loop forever
```

The Global Descriptor Table

Oh, and remember when we promised the hardware that we were gonna give it a GDT? We even told it to load it from address `gdtdesc`, remember? Well, we have to deliver on that promise now by defining the GDT here.

x86 expects that the GDT will be aligned on a 32-bit boundary, so we tell the assembler to do that. Then we use the macros `SEG_NULLASM` and `SEG_ASM` defined in asm.h to create three segments: a null segment, a segment for executable code, and another for writeable data. The null segment has all zeroes; the first argument to `SEG_ASM` has the permission bits, the second is the physical base address, and the third is the maximum virtual address. As we said before, xv6 relies mostly on paging, so we set the segments to go from 0 to 4 GB so they identity-map all the memory.

```
.p2align 2      # force 4-byte alignment
gdt:
    SEG_NULLASM                             # null segment
    SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code segment
    SEG_ASM(STA_W, 0x0, 0xffffffff)         # data segment

gdtdesc:
    .word   (gdtdesc - gdt - 1)     # sizeof(gdt) - 1
    .long   gdt                     # address of gdt
```

bootmain.c

Okay, the rest of the boot loader is in C now! Most of the code here is just to interact with the disk in order to read the kernel from disk and load it into memory. Let's start off by looking at `waitdisk()`.

waitdisk

```
void waitdisk(void)
{
    while ((inb(0x1F7) & 0xC0) != 0x40)
        ;
}
```

HEAD. DESK. Why all the magic numbers? At least we're lucky that the name makes it obvious what this function does; this won't always be true in xv6. Okay, so this function does only one thing: it loops until the disk is ready. Disk specs are boring as all hell, so feel free to skip to the next section if you don't care about the particulars (I don't blame you).

The usual way to talk to the disk is with Direct Memory Access (DMA), in which devices are hooked up directly to RAM for easy communication. But we haven't initialized the disk at all or set up any drivers for it; that's the OS's responsibility, not the boot loader's. Even if we could ask the disk to give us some data through memory-mapped I/O, we disabled all interrupts, so we wouldn't know when it's ready. So instead, we have to go back to assembly code (ugh, I know) to access the disk directly.

Storage disks have all kinds of standardized specifications, among them IDE (Integrated Drive Electronics) and ATA (Advanced Technology Attachment). The ATA specs include a Programmed I/O mode where data can be transferred between the disk and CPU through I/O ports. This is usually a huge waste of resources because every byte has to be transferred through a port and the CPU is busy the entire time, but right now beggars can't be choosers.

Each disk controller chip has two buses (primary and secondary) for use with ATA PIO mode; the primary bus sends data on port 0x1F0 and has control registers on ports 0x1F1 through 0x1F7. In particular, port 0x1F7 is the status port, which will have some flags to let us know what it's up to. The sixth bit (or 0x40 in hex) is the RDY bit, which is set when it's ready to receive more commands. The seventh bit (i.e., 0x80) is the BSY bit, which if set says the disk is busy.

Since interrupts are disabled, we'll have to manually poll the status port in an infinite loop until the BSY bit is not set but the RDY bit is: `inb()` is a C wrapper (defined in x86.h) for the x86 assembly instruction `inb`, which reads from a port. We don't care about any of the other status flags, so we'll get rid of them by bitwise-ANDing the result with 0xC0 = 0x40 + 0x80. If the result of that is 0x40, then only the RDY bit is set and we're good to go.

Phew. That was a lot for just one line of code.

readsect

```
void readsect(void *dst, uint offset)
{
    // Issue command
    waitdisk();
    outb(0x1F2, 1);
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20);

    // Read data
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}
```

If you skipped the last section: this function reads a sector (which in the current-year-according-to-xv6 of 1995 is 512 bytes) from disk. Good to see you again, on to the next section for you!

If you powered through the pain and read about ATA PIO mode above, some of the magic numbers here might be familiar. First we call `waitdisk()` to wait for the RDY bit, then we send some stuff over ports 0x1F2 through 0x1F7, which we know are the command registers for the primary ATA bus.

Note that `uint` is just a type alias for C's `unsigned int`, defined in the header file types.h. The `offset` argument is in bytes, and determines which sector we're gonna read; sector 0 has to hold the boot loader so the BIOS can find it, and in xv6 the kernel will start on disk at sector 1.

`outb()` is another C wrapper for an x86 instruction from x86.h; this one's the opposite of `inb()` because it sends data out to a port. The disk controller register at port 0x1F2 determines how many sectors we're gonna read. Ports 0x1F3 through 0x1F6 are where the sector's address goes. If you *really* must know (why?) they're the sector number register, the cylinder low and high registers, and the drive/head register, in order. Port 0x1F7 was the status port above, but it also doubles as the command register; we send it command 0x20, aka READ SECTORS.

Then we wait for the RDY bit again before reading from the bus's data register at port 0x1F0, into the address pointed to by `dst`. Once again, `insl()` is a C wrapper for the x86 instruction `insl`, which reads from a port into a string. The `l` at the end means it reads one long-word (32 bits) at a time.

readseg

```c
void readseg(uchar *pa, uint count, uint offset)
{
    uchar *epa = pa + count;

    // Round down to sector boundary
    pa -= offset % SECTSIZE;

    // Translate from bytes to sectors; kernel starts at sector 1
    offset = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time. We'd write
    // more to memory than asked, but it doesn't matter -- we load in increasing
    // order.
    for (; pa < epa; pa += SECTSIZE, offset++) {
        readsect(pa, offset);
    }
}
```

Okay, finally, we're done with assembly and disk specs. We're gonna read `count` bytes starting from `offset` into physical address `pa`. Note that `uchar` is another type alias for `unsigned char` from types.h; this means that `pa` is a pointer (which is 32 bits in x86) to some data where each piece is 1 byte.

`epa` will point to the end of the part we want to read. Now, `count` might not be sector-aligned, so we fix that. Declaring `pa` as a `uchar *` lets us do this pointer arithmetic easily because we know that adding 1 to `pa` makes it point at the next byte; if it were a `void *` like in `readsect()`, pointer arithmetic would be undefined. (Actually, GCC lets you do it anyway, but GCC lets you get away with a lot of crazy stuff, so let's not go there.)

Now that we've got everything set up, we just call `readsect()` in a for loop to read one sector at a time, and that's it!

Some people have asked about the structure of some of the for loops in xv6, because they don't always use obvious index variables like `int i`. There are plenty of reasons to hate C, but I think the way it structures for loops is by far one of its most powerful features:

```c
for (initialization; test condition; update statements) {
    code
}
```

When evaluating the for loop, C first executes anything in the initialization. Then it checks whether the test condition is true; if so, it executes the code inside the loop. Then it carries out the update statements before checking the test condition again and runnning the code if it's still true.

In the for loop above, the initialization is just an empty statement; all the variables we want to use have already been set up, so we don't need it and C will just move on to the next step. The test condition is simple enough. But the update statement actually increments both `pa` and `offset` at once before going through the loop again.

Okay great, so now we can read from the disk into memory, so we're all set up to load the kernel and start running it!

ELF Files

Before we move on to the star of the show, `bootmain()`, we need to talk about how a computer can actually recognize a file as executable. When you compile some code, the result gets spit out in a format that your machine can recognize, load into memory, and run; it's usually the linker's job to do this. Most Unix and Unix-like systems use the standardized Executable and Linkable Format, or ELF, for this purpose.

ELF divides the executable file into sections: `text` (the code's instructions), `data` (initialized global variables), `bss` (statically-allocated variables that have been declared but not initialized), `stab` and `stabstr` (debugging symbols and similar info), `rodata` (read-only data, usually stuff like string literals).

An ELF file starts with a header which has a magic number: 0x7F followed by the letters "ELF" represented as ASCII bytes; an OS can use this to recognize an ELF file. The header also tells you the file's type: it could be an executable, or a library to be linked with executables, or something else. There's a whole bunch of other info in the header, like the architecture it's made to run on, version, etc., but we're gonna ignore most of that.

The most important parts of the header are the part where it tells us where in the file the processor should start executing instructions and the part that describes the number of entries, on-disk offset, and size of the program header table.

The program header table is an array that has one entry for each of the file sections above that's found in this program. It describes the offset in the file where each section can be found along with the physical and virtual address at which that section should be loaded into memory and the size of the section, both in the file and in memory; these might differ if, e.g. the program contains some uninitialized variables which don't need to be stored in the file but do need to have space in memory.

The kernel (along with all the user-space programs) will be compiled and linked as ELF files, so `bootmain()` will have to parse the ELF header to find the program header table, then parse that to load each section into memory at the right address. xv6 uses a `struct elfhdr` and a `struct proghdr`, both defined in elf.h, for this purpose.

Okay, back to the boot loader to finish up now!

bootmain

This is the C function that gets called by the first part of the boot loader written in assembly. Its job will be to load the kernel into memory and start running it at its entry point, a program called `entry()`.

Next up, we're gonna use `readseg()` to load the kernel's ELF header into memory at physical address 0x1_0000; the number isn't too important because the header won't be used for long; we just need some scratch space in some unused memory away from the boot loader's code, the stack, and the device memory-mapped I/O region. We'll read 4096 bytes first at offset 0; `readseg()` turns that offset into sector 1. Remember that we have to convert `elf` into a `uchar *` so that the pointer arithmetic in `readseg()` works out the way we want it to.

```c
void bootmain(void)
{
    struct elfhdr *elf = (struct elfhdr *) 0x10000;
    readseg((uchar *) elf, 4096, 0);
    // ...
}
```

While we're at it, let's go ahead and make sure that what we're loading really is an ELF file and not some random other garbage because any of a million things went wrong during the compilation process, or we got some rootkit that totally corrupted the kernel or something. It's not really the most robust of checks, but *eh*. If something went wrong we'll just return, since we know that the code in `bootasm.S` is ready to handle that with some Bochs breakpoints.

```c
void bootmain(void)
{
    // ...
    if (elf->magic != ELF_MAGIC) {
        return;
    }
    // ...
}
```

Now we have to look at the program header table to know where to find each of the kernel's segments. The `elf->phoff` field tells us the program header table's offset from the start of the ELF header, so we'll set `ph` to point to that and `eph` to point to the end of the table.

```c
void bootmain(void)
{
    // ...
    struct proghdr *ph = (struct proghdr *) ((uchar *) elf + elf->phoff);
    struct proghdr *eph = ph + elf->phnum;
    // ...
}
```

Each entry in the program header table tells us where to find a segment, so we'll iterate over the entries, reading each one from disk and loading it up. In this for loop, note that `ph` is a `struct proghdr *`, so incrementing it with `ph++` increments it by the size of a `struct proghdr` and not by one byte; this makes it automatically point at the next entry in the table.

```c
void bootmain(void)
{
    // ...
    for (; ph < eph; ph++) {
        uchar *pa = (uchar *) ph->paddr;    // address to load section into
        readseg(pa, ph->filesz, ph->off);   // read section from disk

        // Check if the segment's size in memory is larger than the file image
        if (ph->memsz > ph->filesz) {
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
        }
    }
    // ...
}
```

That if statement at the end checks if the section's size in memory should be larger than its size in the file, in which case it calls `stosb()`, which is yet another C wrapper from [x86.h](x86.h) for the x86 instruction `rep stosb`, which block loads bytes into a string. It's used here to zero the rest of the memory space for that section. Okay, but why would we want to do that? Well, if the reason it's larger is because it has some uninitialized static variables, then we want to make sure those start off holding zero (as the C standard requires) and not whatever garbage value may have been there before.

Last part of the bootloader: let's call the kernel's entry point, `entry()`, and get it running! But remember how the boot loader is compiled and linked separately from the kernel? Yeah, that means we can't just call `entry()` as a function, because then the linker would go "Huh? What entry function? I don't have any `entry` function here in your symbol table. REJECTED." And then it would throw a huge error.

Luckily, the ELF header tells us where to find the entry point in memory, so we could get a pointer to that address. That means... function pointers! If you've never used function pointers in C before, then this won't be the last time you'll see them in xv6, so check it out.

A C function is just a bunch of code to be executed in order, right? That means it shows up in the ELF file's `text` section, which will end up in memory. When you call a regular old C function, the compiler just adds some extra assembly instructions to throw a return address on the stack and update the registers `%ebp` and `%esp` to point to the new function's stack on top of the old one. If the function getting called has any arguments or local variables, they'll get pushed onto the stack too. Then the instruction register `%eip` gets updated to point to the new function section, and that's it. After the compiler is done, the linker will replace the function's name with its memory address in the `text` section, and voila, a function call.

The point of all this is that in C we can use pointers to functions; they just point to the beginning of that function's instructions in memory, where the `%eip` register would end up pointing if the function gets called. So in this case, even though we're not linking with the kernel, we can still call into the entry point by getting its address from the ELF header, creating a function pointer to that address, then calling the function pointer. The compiler will still add all the usual stack magic, but instead of the linker determining where `%eip` should point, we'll do that ourselves.

The first line below declares `entry` as a pointer to a function with argument type `void` and return type `void`. Then we set `entry` to the address from the ELF header, then we call it.

Again, this shouldn't return, but if it does then it's the last part of this function, so this function will return back into the assembly boot loader code.

```c
void bootmain(void)
{
    // ...
    void (*entry)(void);
    entry = (void(*) (void)) (elf->entry);
    entry();
}
```

That's it! Starting from `entry()`, we're officially out of the boot loader and into the kernel.

Summary

To summarize, the assembly part of the boot loader (1) disabled interrupts, (2) set up the GDT and segment registers so the segmentation hardware is happy and we can ignore it later, (3) set up a stack, and (4) got us from 16-bit real mode to 32-bit protected mode.

Then the C part of the boot loader just loaded the kernel from disk and called into its entry point.

ELF headers will continue to haunt us in the kernel's linker script and when we load user programs from disk in `exec()`, and function pointers will make another appearance when we get around to handling interrupts. The good news: the boot loader is one of the most opaque parts of the xv6 code, full of boring hardware specs and backwards-compatibility requirements, so if you made it this far, it does get better!

(But it also gets worse... looking at you, mp.c and kbd.c...)