The workings of IA32 real mode addressing and the a20 line

by Perica Senjak

Preface

I have written this document to explain how real mode addressing works on IA32 compatible CPUs (hereon referred to as just IA32 CPUs) and what the "a20 line" (which is also sometimes called the "a20 gate"; in my opinion though both names do not suit the "phenomenon" they are referring to, but that's off the point; this text will refer to it as the "a20 line") is and how it works on IBM compatible PCs (hereon referred to as IBMCPCs).

Background knowledge about computers and computer architectures is assumed, this text is only meant to explain a very small part (real mode addressing and the a20 line) of the rather weird and unique (and horrendous, for reasons that I will not go into; and also great for reasons that I will not go into either) IBMCPCs.

This text is written with the assumption that you are writing a program that directly boots off an IBMCPC (for example, something like an operating system loader). This entire text is talking about the IBMCPC at the state it is at boot up.

For further reading I highly suggest you search for the reference manuals of the processor that you are programming for, these can be easily found on the internet and downloaded at no cost.

Real mode and real mode addressing

IA32 CPUs are able to run in two addressing modes, real mode (which is a 16-bit unprotected addressing mode) and protected mode (which is a 32-bit protected addressing mode). Memory addressing is handled very differently in protected mode as opposed to real mode, but nothing about protected mode and the differences it has with real mode will be discussed in this text as it is beyond its scope.

In IBMCPCs the address space mainly maps RAM (**r**andom **a**ccess **m**emory) but it also maps VRAM (**v**ideo **r**andom **a**ccess **m**emory), the BIOS's ROM (**b**asic **i**nput and **o**utput **s**ystems **r**ead **o**nly **m**emory) chips and various other things. Real mode can address just over 1 megabyte of address space (1,114,096 bytes to be exact). The address space at 1 megabyte and above can not be accessed while the a20 line is disabled though (most IBMCPCs boot with the a20 line disabled; if you want to access the address space at 1 megabyte and above then you must check if the a20 line is enabled and if not, you must enable it). The a20 line will be discussed in more detail later on in this text.

The state (what devices are mapped into the address space, etc.) of the IBMCPC's first 1,114,096 bytes of physical address space at boot-up is like this (this may not make sense at the moment, since you probably do not understand real mode addressing, but it will be useful later on so you might want to skip over this memory map and refer back to it later on when necessary):

0x0000:0x0000 -> 0x0000:0x03FF = Real mode IVT (**i**nterrupt **v**ector **t**able).
0x0000:0x0400 -> 0x0000:0x0FFF = The BDA (**B**IOS **d**ata **a**rea) and various other memory that's sometimes used by the BIOS that's best left alone.
0x0000:0x1000 -> 0x0000:0x7BFF = Free memory.
0x0000:0x7C00 -> 0x0000:0x7DFF = 512 byte boot sector of the device that was booted from, this memory may be considered as free memory as long as the boot sector isn't executing and any data contained in the boot sector isn't needed anymore.
0x0000:0x7E00 -> 0x9000:0xFFFF = Free memory.

0xA000:0x0000 -> 0xB000:0xFFFF = VGA (**v**ideo **g**raphics **a**rray) VRAM.
0xC000:0x0000 -> 0xF000:0xFFFF = The BIOS's ROM chips.
0xFFFF:0x0010 -> 0xFFFF:0xFFFF = Free memory (which may only be used if the a20 line is enabled; this will be discussed in detail later).

Important note on real mode memory map: more devices and important structures such as the EBDA (**e**xtended **B**IOS **d**ata **a**rea) may be mapped into or present in the physical address space (although unlikely), as not all IBMCPCs are completely compatible with each other; it is highly recommended that you get a memory map from the BIOS at run-time (where possible, as not all BIOS's support it). How to get a memory map from the BIOS is beyond the scope of this text though, I recommend you do a search on the internet for memory map BIOS functions on IBMCPCs.

All early IBMCPCs were 16-bit machines which meant that they also had a 16-bit address bus. The designers of the early IBMCPCs were confronted with a problem, by the limitations of the address bus their machines would only be able to address up to 64 kilobytes of memory - this was way too little memory for the requirements at the time (there needed to be enough address space to map in VRAM, the BIOS's ROM chips, etc.). The solution to the problem was, instead of sending one 16-bit value over the address bus, two 16-bit values were sent over the address bus to make up the final physical address. The first value is called the segment and the second value is called the offset. The two 16-bit values are translated using the formula *(segment << 4) + offset* to get the final physical address to be used. Whenever your code is in real mode, just remember these address translation rules that the processor does. Simple as that.

The a20 line

The address bus in the early IBMCPCs was only capable of addressing up to 1 megabyte of physical address space, but the addressing scheme they used was capable of addressing up to just over 1 megabyte. So what they did was wrap around the spare address space from 0xFFFF:0x0010 and above to 0x0000:0x0000 and above (so addressing 0xFFFF:0x0010 (which is equal to physical address 0x00100000) would be the same as addressing 0x0000:0x0000, addressing 0xFFFF:0x0011 would be the same as addressing 0x0000:0x0001, etc.). But as technology developed, the address bus of IBMCPCs was capable of addressing more than 1 megabyte of physical address space. This was all well and good, but there was a lot of software out there that depended on the memory wrap around of the older IBMCPCs to function properly. Compatibility was a big deal back then, so they created something called the a20 line to ensure compatibility with older machines. The a20 line is a switch that enables and disables the memory wrap around, the idea was that the computer would boot with it disabled so all older software would be compatible with the newer IBMCPCs and any software that wanted to take advantage of the extra memory would enable the a20 line to disable the memory wrap around.

In today's world of modern computing, the a20 line is just a nuisance so you most likely want it enabled. Unfortunately, the a20 line is a compatibility issue in itself as not all IBMCPCs have one universal way to enable it. So the only option to maintain maximum compatibility is to write code that attempts multiple ways to enable the a20 line.

Below is code for one way to enable the a20 line, an explanation of what it's doing is not provided as it goes beyond the scope of this text - this is not the only way to enable the a20 line, there are many ways but this is the one that has compatibility with most machines; you'll have to do a little bit of extra research yourself to find out what's going on and to write code to check if the a20 line has been enabled successfully and if not get it to try other ways to enable it. The below code uses what is commonly referred to as the "keyboard controller method of enabling the a20 line".

```
; ********** START OF CODE LISTING **********

; This code is written in Intel assembler; written to be assembled
```

```
using the NASM assembler.
; This code is in the public domain - do absoloutely anything you want
with it.

      jmp enable_a20_line

 keyboard_input_buffer_wait:
      in al, 0x64
      and al, 0x02
      cmp al, 0
      jne keyboard_input_buffer_wait
      retn

 keyboard_output_buffer_wait:
      in al, 0x64
      and al, 0x01
      cmp al, 0
      jne keyboard_output_buffer_wait
      retn

 enable_a20_line:
      cli
      call keyboard_input_buffer_wait
      mov al, 0xD0
      out 0x64, al
      call keyboard_output_buffer_wait
      in al, 0x60
      mov ah, al
      or ah, 0x02
      call keyboard_input_buffer_wait
      mov al, 0xD1
      out 0x64, al
      call keyboard_input_buffer_wait
      mov al, ah
      out 0x60, al
      call keyboard_input_buffer_wait
      sti
      retn
; ********** END OF CODE LISTING **********
```