The Workings of: x86-16/32 RealMode Addressing

by Perica Senjak

Preface

I have written this document to explain in "Detail" how RealMode Addressing works (And Other things Related to it!); The Reason i wrote this is because when i first Started Operating System Development (Which is Not so long ago) It took me quite some time to find out how RealMode Addressing Works, the reason is because there weren't any good (free) Documents based on it (And i don't think there are any good one's now); So, i have decided to make it easy for you and write this tutorial :)

In this tutorial you will learn Everything you need to know about RealMode Addressing, we will Start with the Absoloute Basics - then move up to more advanced things! For this tutorial to be usefull to you, you must know some Assembler Terms (Assembler is a Low-Level Programming Language);

If you are not an Operating System Developer, or if you are not developing on the x86 Platform - Then this tutorial will probably be useless to you: if this is your case, then i recommend you don't waste your time here - But feel free to read it anyway if you want to, you never know, you might learn something ;)

That's Enough talk, Lets get Down to Business.....

What is RealMode and RealMode Addressing?

The x86-16/32 CPU Runs in Two Modes - RealMode (*16-Bit*) and ProtectedMode (*32-Bit*), (There are a few big Differences Between these modes - The *HUGE* Difference is the way Memory is handled - We will not explain the other differences: This information is beyond the purpose of this tutorial, We will also not explain anything about ProtectedMode in this tutorial - As this tutorial is all about RealMode.)
NOTE: On the x86 Platform, the CPU (Central Processing Unit) Boots into RealMode.

On the x86 Platform there is something called Address Space, This is space that maps out Memory on various devices - It mostly maps out RAM (Randon Acess Memory - The Computer's Main Memory) But it also maps other Memory such as: Video Memory (Video RAM), The BIOS ROM's (Read Only Memory) - There are some other things aswell. In RealMode you can only acess 1mb of Address Space - This 1mb Maps out RAM, Video RAM, and the BIOS ROM's - *Some of the memory is also taken up.*

Below is a Memory-Map of RealMode Memory, it shows what is located at each Memory Address (In RealMode) - It also shows if that Memory Address is free for system use (When i say "System Use" i am referring to the State of the Memory when the System "Boots" - You might not Understand what this Map means right now - But it will help later on, So you might want to come back to it later);

```
0x0000:0x0000 -> 0x0000:0x03FF = IVT (Interrupt Vector Table)
0x0000:0x0400 -> 0x0000:0x04FF = BDA (BIOS Data Area)

0x0000:0x0500 -> 0x0000:0x7BFF = Free Useable Memory!

0x0000:0x7C00 -> 0x0000:0x7DFF = Operating System BootSector - This is where the BIOS
Loads your Operating System's BootSector at Boot Time (You can use this Memory, as long as
your BootSector isn't executing and you don't need your BootSector anymore!)

0x0000:0x7E00 -> 0x9000:0xFFFF = Free Useable Memory!
```

```
0xA000:0x0000 -> 0xB000:0xFFFF = VGA Video RAM
0xC000:0x0000 -> 0xF000:0xFFFF = BIOS ROM Memory Area

0xFFFF:0x0010 -> 0xFFFF:0xFFFF = Free Useable Memory (If the A20 Gate is Enabled) - This
Memory is Above the 1mb Mark, 0xFFFF:0x0010 = 0x00100000(1mb)
```

In RealMode (And in a way - This also Applies to ProtectedMode), Every Memory Location has a "Addressable Address" and a "Absoloute Address" -- The "Addressable Address" is the *Segment:Offset* pair (In ProtectedMode it's the *Selector:Offset* pair), the "Absoloute Address" on the other hand is the Actual Memory Location that is Refered to by the "Addressable Address." The Addressable Address is just a way to Refer to the Absoloute Address -- The Reason it Exists is because most Absoloute Addresses are too big to transfer over the System BUS (*This is Explained in more Detail later on...*), So the Address is sent in two parts- A Segment:Offset pair (*NOTE: This is not the Case in ProtectedMode - ProtectedMode works Completely Differently!*)

This may seem a Bit unclear now, Hopefully you will understand it better by the time you finish this tutorial, Remember: If you can't understand something, Read it Over-and-Over again until you do!

How does RealMode Memory work and How Can i acess it?

Alright, this is one of the most Complicated parts of RealMode and the x86 Architecture, But it is also one of the most important -- This will take some time to Explain, So please Bear with me and Remember: If you don't understand something read it over and over again until you do!, Lets Start...

RealMode Memory is acessed through a *Segment:Offset* pair - The *Segment* is a *Memory Base*, and the *Offset* is the *Location from the Base*. *Each Segment can address up to 0xFFFF (65535) Bytes of memory.* As i said Earlier: In RealMode the BUS can only Transfer a Limited Value (*Meaning: There is a Size Limit to what the System BUS can transfer, This limit is way too small in RealMode*), The Limit was a big problem - If the memory was referred to by it's Absoloute Address (*Absoloute Addresses are Explained Above, They are also Explained a Bit Better below - For those of you who still don't fully understand*) then only a Small Number of Bytes of Memory Could be Acessed - So they invented the *Segment:Offset* Addressing. The Memory Address gets sent in two parts - A *Segment:Offset* pair (*Therefore they could transfer up to a 20-Bit value, **With the A20-Gate as an Exception - The A20-Gate Allows you to Acess Upper-Memory: The A20-Gate is Explained in Detail in the Next Section of this Tutorial - Don't worry about it for now**);*

Absoloute Addresses start at Memory Location *0x00000* and they just keep on increasing by one, so *0x00000* is the First Byte of System Memory - *0x00001* is the Second and so forth. The *Segment:Offset* is just a way to Refer to memory: *There is no fixed number of Segments!! -- It is Just a way to Refer to Memory!*

The *Segment:Offset* System Refers to an *Absoloute Address* (*As i have Explained Above*); The Formula to Calculate the *Absoloute Address* that the *Segment:Offset* pair is Pointing to is Easy, You *use this formula: **"Segment * 16 + Offset"** - You multiply the Segment by 16 and add the Offset to it - This will Calculate the Absoloute Address.* The Formula to turn an *Absoloute Address* into a *Segment Address* is: ***"Absoloute Address / 16"** - You divide the Address by 16; **NOTE:*** The *Absoloute Address MUST* end with a 0, otherwise you can not divide it evenly (*You will get a decimal value*) -- *You can figure out exactly why it won't work youself: It isn't Rocket Science!! -- The Absoloute Address also must be under 0x000FFFF0 to be used as a Segment.*

There is also another thing to Consider in the *Segment:Offset* Addressing System - It is *Segment Overlap*. For Example: If you wanted to acess the Memory Location *0x07C00* (*Absoloute Address*) You could do it in two

ways, you could do it using this Address: *0x0000:0x7C00*, or using this Address: *0x07C0:0x0000 - Both of these Addresses point to the Same Absoloute Address*, This is all that Segment Overlap is! You can Check for youself - Just Calculate the *Absoloute Address* of both of these *Segment:Offset* Addresses, you will notice that they both refer to Absoloute Address 0x07C00! ...That's all that *Segment Overlap* is, it is not a problem - You just should be aware of it, and explaining *Segment Overlap* should help you understand the way that *RealMode Addressing* works!

There, that is Everything - You should now Understand RealMode Addressing, if you don't: Read over it again until you do!, It is quite complicated, so keep on Reading it over and over until you understand!

Below the A20-Gate is Explained, it is a Very Important Part of the x86 Architecture - Please Read these Parts of the tutorial aswell -- To fully understand x86 RealMode Addressing: You MUST understand the A20-Gate aswell!! Read On...

What is the A20 Gate?

The A20-Gate (Also Called the "A20-Line") is a way to Eliminate Memory Wrap-Around, What is Memory Wrap-Around you ask? Well, in the Early Days of the x86 Architecture the Machine was Limited to only 1mb of Addressable Memory, although it had the Capability of 1mb of Address Space it only had Just under 1mb of RAM (20 Address Lines) - So the Machine was left with Spare Address Space (*0xFFFF:0x0010* -> *0xFFFF:0xFFFF*). So, What did they do with this Spare Address Space you ask? Well, they made the Memory Wrap-Around (It Wrapped-Around to: *0x0000:0x0000* and up)!!

Memory Wrap-Around does exactly what the name suggests, it makes the memory wrap around! So, for example - If you tried to Acess the Address *0xFFFF:0x0010* you would actually be acessing the Memory at *0x0000:0x0000*.

Later on, when computers advanced - They were capable of more RAM, the Memory Wrap-Around was going to be Removed, and the extra RAM was going to be mapped at the Unused Address Space. But there was a problem: Some old Programs Relied on the Memory Wrap-Around to function properly! So they invented a thing called the A20-Gate - The A20-Gate was Disabled by Default, and if a Program wanted to Acess the Extra Memory, they would simply Enable the A20-Gate - Even today the A20-Gate has to be Enabled - It is Very Important, Even in ProtectedMode -- If you want to use the full Functionality of ProtectedMode: The A20-Gate *Must* be Enabled!!

How do i Enable the A20-Gate?

There are multiple ways to Enable the A20-Gate, I will show you the two most Common ways (Through the Keyboard Controller and Through Port 0x92) - There are other ways, what i recommend you do is: Try and Enable the A20-Gate through the Keyboard Controller first, then check if it Enabled Properly (Read Below on how to do this), if it doesn't enable Properly - Then try doing it through Port 0x92, I Recommend you Have about 3 Back-Up ways of Enabling the A20-Gate (Just in Case), Enabling the A20-Gate is a very important part in your Operating System's BootProcess!

Below is the Source-Code to Enable the A20-Gate through Port 0x92 - This is the Simplest way, But will only work on Newer Machines - That's why you will have Two Back-Up ways (Never have only One way of Enabling the A20-Gate!!), Here is the Code (There is no need for me to Explain it, It is to some extent Self-Explanatory):

```
cli

in al, 0x92
or al, 0x02
out 0x92, al

sti
```

And here is the Source-Code for the Keyboard Controller way of Enabling the A20-Gate, this Source-Code is also Self-Explanatory to Some Extent, So i will not Explain it:

```
 jmp Enable_A20Gate

 Empty_KeyboardBuffer:
                xor al, al
                in  al, 0x64

                test al, 0x02
                jnz Empty_KeyboardBuffer

                ret

 Enable_A20Gate:
                cli

                call Empty_KeyboardBuffer

                mov al, 0xD1
                out 0x64, al

                call Empty_KeyboardBuffer

                mov al, 0xDF
                out 0x60, al

                call Empty_KeyboardBuffer

                sti
```

There, those are the Two Most Common ways of Enabling the A20-Gate; As i said: There are many other ways, so *be Sure* to have two or Three Lined up Just in Case One Fails!

How do i Check if the A20 Gate Has Been Enabled Properly?

After you Enable the A20-Gate (Or Should i say "Try and Enable the A20-Gate") you will want to check if it was Enabled properly, if it wasn't - then you can try another method, as you know there are many ways to Enable the A20-Gate!

Fortunately, it is very easy to "Check" if the A20-Gate is Enabled/Has been Enabled Properly. The A20-Gate Eliminates Memory Wrap-Around (As you know) so if the A20-Gate is NOT Enabled, the memory Wrap-Around will still happen - this is the way you check if the A20-Gate has been Enabled! What you do is: You designate 1 Byte of Memory in the A20-Gate's Wrap-Around Range and you Set this Memory to NULL (Move 0 into it); Then you Enable the A20-Gate, to check if it was Enabled Properly: You move the value 1 into the memory where the Memory Used to Wrap-Around, Just make Sure you move it to the right place - so it is aligned with the Memory you designated, then you Simply Compare the Memory you designated with the value 0 (NULL), If the value's are equal - Then the A20-Gate Has been Enabled Sucessfully, If not - Then the A20-Gate has not been Enabled, then you can use another way to try and Enable the A20-Gate, Just keep doing this until the A20-Gate is Enabled!

Here is some Example Assembler Code on how to Check if the A20-Gate has been Enabled Properly (Note:

This Is RealMode Code - It is Not Commented, It is Self-Explanatory if you Read the above Description - You may need to Edit it to Comply with your Operating System):

```
 xor ax, ax
 mov bx, 0xFFFF
 mov fs, ax
 mov gs, bx

 mov di, 0x0500
 mov si, 0x0510

 mov byte[FS:DI], 0x00

;.....Enable the A20-Gate Here

 mov byte[GS:SI], 0x01
 cmp byte[FS:DI], 0x01

 je Error ;If the A20-Gate does not Get Enabled properly, this Jumps to The Next-Method
          ;or Error Handling Code - It's up to you, Just Replace "Error" with the Place
          ;you wish to Jump if the A20-Gate Does Not Get Enabled Properly!

;.....If the Code Gets to Here, The A20-Gate Has Been Enabled!
```

That Wraps it up!, Hopefully now you have a better understanding of x86 RealMode Addressing, and the x86 Architecture!! Good Luck (For whatever Reason you have Read this Tutorial...)
-Perica Senjak