

# Removing the Mystery from SEGMENT : OFFSET Addressing

Copyright©2001, 2007 by Daniel B. Sedory

**This page may be freely copied for PERSONAL use ONLY !  
( It may NOT be used for ANY other purpose unless you have  
first [contacted](#) and received permission from the author! )**

For information on using MS-DEBUG, see [A Guide to DEBUG](#).

---

- Section 1. [Introduction, Definitions and Statistics](#)
  - Section 2. [Vizualizing the Overlapping Segments](#)
    - [The Upper Memory Area \(UMA\)](#)
    - [The High Memory Area \(HMA\)](#)
  - Section 3. [Normalized Segment:Offset Notation](#)
  - Section 4. [Problems when Segment Register Values are not considered !](#)
- 

## Introduction.

There are often many different Segment:Offset pairs which can be used to address the same location in your computer's memory. This scheme is a **relative way** of viewing computer memory as opposed to a **Linear** or **Absolute** addressing scheme. When an Absolute addressing scheme is used, each memory location has its own *unique* designation; which is a much easier way for *people* to view things. So, why did anyone ever create this awkward "Segment:Offset scheme" for dealing with computer memory? As an answer, here's a brief lesson on the 8086 CPU with an *historical slant*:

Segment:Offset addressing was introduced at a time when the largest register in a CPU was only 16-bits long which meant it could address only 65,536 bytes (64 KiB<sup>[1]</sup>) of memory, directly. But everyone was hungry for a way to run much larger programs! Rather than create a CPU with larger register sizes (as some CPU manufacturers had done), the designers at Intel decided to keep the 16-bit registers for their new

8086 CPU and added a different way to access more memory: They expanded the instruction set, so programs could tell the CPU to *group* two 16-bit registers together whenever they needed to refer to an Absolute memory location beyond 64 KiB.

If the designers had allowed the CPU to combine two registers into a high and low pair of 32-bits, it could have referenced up to 4 GiB<sup>[2]</sup> of memory in a linear fashion! Keep in mind, however, this was at a time when many never dreamed we'd need a PC with more than 640 KiB of memory for user applications and data!<sup>[3]</sup> So, instead of dealing with whatever problems a linear addressing scheme of 32-bits would have produced, they created the Segment:Offset scheme which allows a CPU to effectively address about 1 MiB of memory.<sup>[4]</sup>

The scheme works like this: The value in any register considered to be a Segment register is multiplied by 16 (or shifted one hexadecimal byte to the left; add an extra 0 to the end of the hex number) and then the value in an Offset register is added to it. So, the Absolute address for any combination of Segment and Offset pairs is found by using the formula:

$\begin{array}{l} \text{Absolute} \\ \text{Memory} \\ \text{Location} \end{array} = (\text{Segment value} * 16) + \text{Offset value}$
--

After working through some examples, this will become much clearer to understand: The Absolute or Linear address for the Segment:Offset pair, **F000:FFFD** can be computed quite easily in your mind by simply inserting a zero at the end of the Segment value ( which is the same as multiplying by 16 ) and then adding the Offset value:

$$\begin{array}{r} \text{F0000} \\ + \text{FFFD} \\ \hline \text{FFFFD} \end{array} \quad \text{or} \quad 1,048,573_{(\text{decimal})}$$

Here's another example: 923F:E2FF ->

$$\begin{array}{r} \text{923F0} \\ + \text{E2FF} \\ \hline \text{A06EF} \end{array} \quad \text{or} \quad 657,135_{(\text{decimal})}$$

Now let's compute the Absolute Memory location for the largest value that can be expressed using a Segment:Offset reference:

$$\text{FFFF0}$$

$$\begin{array}{r}
 + \quad \text{FFFF} \\
 \text{-----} \\
 10\text{FFEF} \quad \text{or} \quad 1,114,095 \quad (\text{decimal})
 \end{array}$$

In reality, it wasn't until quite some time after the 8086, that such a large value actually corresponded to a real Memory location. Once it became common for PCs to have over 1MiB of memory, programmers developed ways to use it to their advantage and this last byte became part of what's now called the HMA (High Memory Area). But until that time, if a program tried to use a Segment:Offset pair that exceeded a 20-bit Absolute address (1MiB), the CPU would *truncate* the highest bit (an 8086/8088 CPU has only 20 address lines), effectively *mapping* any value over **FFFFh** (1,048,575) to an address within the first Segment. Thus, 10FFEFh was mapped to FFEFh. [\[5\]](#)

One of the *downsides* in using Segment:Offset pairs (and likely what confuses most of you) is the fact that a large number of these *pairs* refer to the same exact memory locations. For example, every Segment:Offset *pair* below, refers to ***exactly the same location*** in memory:

0007:7B90	0008:7B80	0009:7B70	000A:7B60	000B:7B50	000C:7B40
0047:7790	0048:7780	0049:7770	004A:7760	004B:7750	004C:7740
0077:7490	0078:7480	0079:7470	007A:7460	007B:7450	007C:7440
01FF:5C10	0200:5C00	0201:5BF0	0202:5BE0	0203:5BD0	0204:5BC0
07BB:0050	07BC:0040	07BD:0030	07BE:0020	07BF:0010	07C0:0000

**The Segment:Offset pairs listed above are only some of the many ways one can refer to the single Absolute Memory location of: 7C00h (or 0000:7C00).**

As a matter of fact there *may be* up to **4,096** different Segment:Offset pairs for addressing a single byte in Memory; depending upon its particular location. For Absolute addresses **0h** through **FFFFh** ( 0 through 65,519 ), the number of different pairs can be computed as follows: Divide the Absolute address by 16 ( which shifts all the hex digits one place to the right ), then *throw away* any fractional remainder and add 1. This is the same thing as saying: Add 1 to the Segment number if the Offset is 000Fh (15) or less. For example, the byte in memory referenced by the Segment:Offset pair **0040:0000** has a total of 41h (or 65) different pairs that might be used. For the Absolute address **7C00h**, which was mentioned above, there's a total of:  $7C00 / 10h \rightarrow 7C0 + 1 = 7C1$  (or 1,985) *relative* ways to address this same memory location using Segment:Offset pairs. For the Absolute addresses from **FFF0h** (65,520) all the way through **FFFFh** (1,048,575), there will always be **4,096 Segment:Offset pairs** one could use to refer to these addresses! That's a little over **88%** of the memory that can be accessed using Segment:Offsets. The last 65,520 bytes that can be accessed by this method are collectively called the High Memory Area (HMA). For each 16 bytes higher in the **HMA** that we point to, there is

one less Segment:Offset pair available to reference that paragraph.

Due to the sheer number of possible Segment:Offset pairs for each address, most programmers have agreed to use the same *normalization method* (see the note below on [Normalized Notation](#)) when writing about a particular location in Memory.

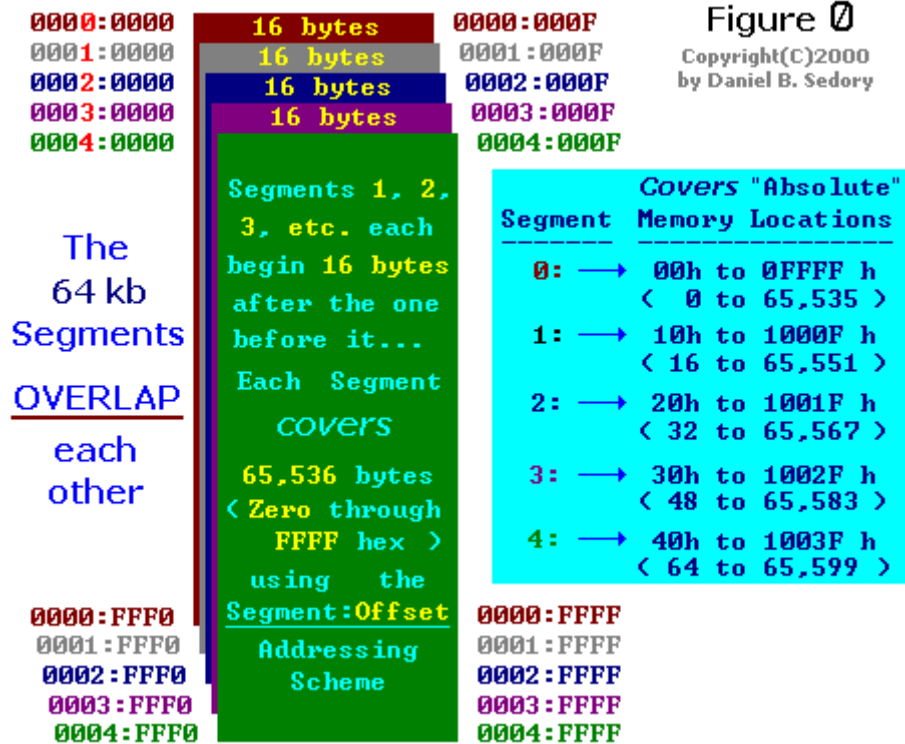
We've created some graphic illustrations to help you picture the boundaries between various areas of Memory:

---

# Visualizing the Overlapping Segments

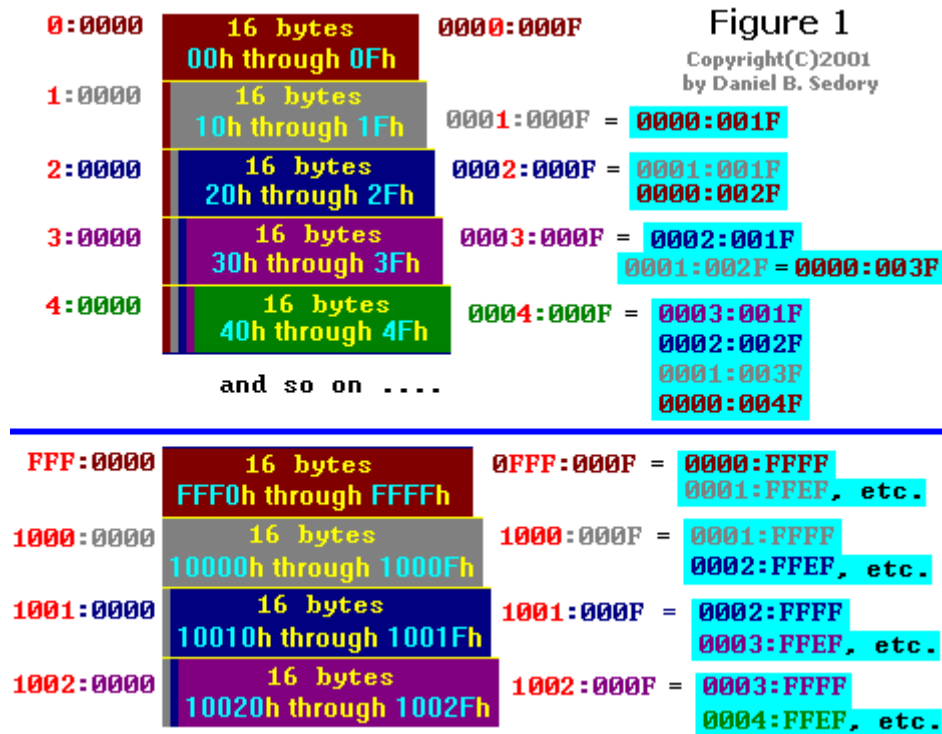
**The following illustrations should help students *visualize the artificial layout* of the Segment boundaries in a system's Memory.**

SEGMENTS are more like a mental construct or a way of visualizing a computer's Memory, rather than being closely tied to the physical hardware itself. In **Figure 0**, we've tried to show how each *Segment* of 65,536 bytes ***overlaps most of*** the preceding Segment. As you can see, each Segment begins only 16 bytes (or a paragraph) after the preceding one. In computer terminology, a ***paragraph*** is used to refer to 16 consecutive bytes of Memory. For every 16 bytes higher in Memory that we point to, the number of overlapping Segments will increase by one **until** we arrive at the end of the first Segment. At that point, each successive paragraph of Memory (up to 1MiB) has a constant number of 4,096 overlapping Segments! Figure 0 also shows the Segment:Offset values for each of the four *corners* of the first five of Segments.



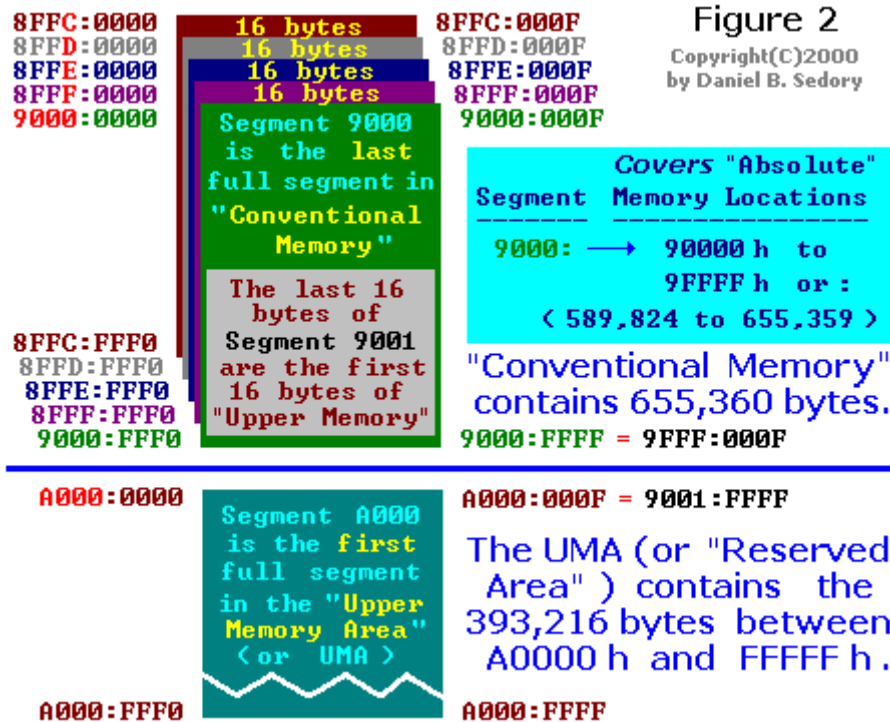
In Figure 1 (see below), the focus is on just the beginning of Segments 1, 2, 3 and so on, rather than the whole Segment. Notice how the first 16 bytes of memory appear in the Figure. There's only one segment there: *no other segments overlap* these bytes. Therefore, the Segment:Offset pairs for each of the first 16 bytes in memory is actually unique! There's only one way to refer to them: with the Segment value of 0000: and one of the 16 Offsets, 0000 through 000F hex. The next 16 bytes in memory (10h through 1Fh) will each have precisely **two** different Segment:Offset pairs that can be used to address them. For each of the first five Segments, the exact number of equivalent Segment:Offset pairs for the last byte in the paragraph has been shown in the aqua (light-blue) colored boxes.

(For comments on the part of Figure 1 under the BLUE line, see text below.)



The second part of Figure 1 above, shows what happens at the transition from a paragraph of memory that is still within the *boundary* of the first 64kb Segment (Absolute addresses FFF0h through FFFFh) to those paragraphs which are beyond its boundary (10000h and following). Note that the first paragraph of Segment 0FFF: (which is the same as the last 16 bytes within Segment 0000:) is the first paragraph in Memory to have a total of 4,096 different Segment:Offset pairs that could be used to reference its bytes.

Figure 2 shows that **Segment 9000:** is the last whole 64kb segment to lie within the bounds of what's called " **Conventional Memory** " ( the first 640kb or 655,360 bytes). The first paragraph of **Segment A000:** is the beginning of the Upper Memory Area (**UMA**). The UMA contains a total of 384kb or 393,216 bytes. **Segment F000:** is the last whole segment that lies within the bounds of the UMA.



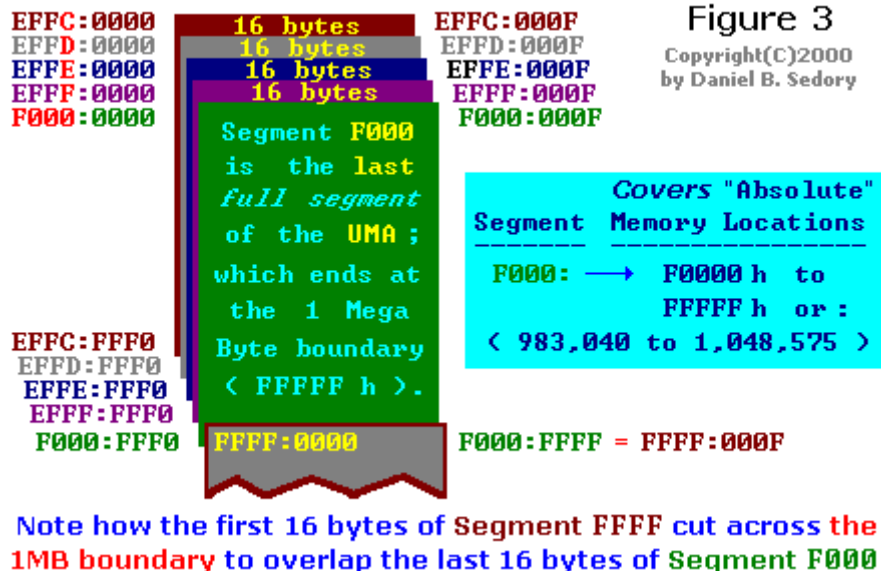
640 KiB + 384 KiB = 1024 KiB (or 1,048,576 bytes) = 1 Mebibyte.

(A long time ago, the UMA was called the 'Reserved Area'.)

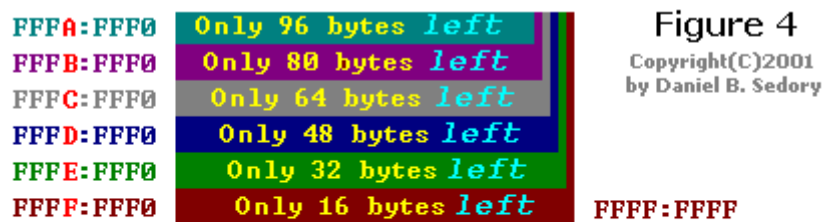
Another way of looking at the first 1MiB of Memory (and hopefully something that will help those who might still be a bit confused ) is the fact that each of these 1,048,576 bytes can be accessed by using one of just the following **16 Segment** references ( **none of which overlap** any of the others): **0000:**, **1000:**, **2000:**, ... **9000:** and **A000:** through **F000:** plus one of their 65,536 Offsets. Although it would really be nice if we could always refer to a particular byte in Memory using just these 16 Segments, that would be rather wasteful of memory resources: When it comes time for an OS like Windows to assign a full 64kb of free memory to an App such as DEBUG, it's not simply a matter of convenience for it to use the very first 16-byte Segment reference of continuous memory that it can find. Moving up to the next even 1000h Segment would leave even more unused **holes** in Memory than there are already! (There is, however, an agreed upon convention called [Normalized Addressing](#) which has been very helpful.)

Figure 3 shows the end of the **UMA** and the beginning of the last Segment (**Segment FFFF:**) in the Segment:Offset scheme. When the 8086 was first created, there wasn't even 640kb of memory in most PCs. And as you might recall from our history lesson above, addresses in this part of the Segment:Offset scheme were first *mapped* to bytes in Segment 0000. Later, the memory above 1MiB that could still be accessed using Segment:Offset pairs became known as The High Memory Area (**HMA**).





The High Memory Area (HMA) contains **only one paragraph** short of 64kb (or just 65,520 bytes). **Segment FFFF:** is the only segment that can *reach* the last 16 bytes of the HMA. Here's a text file of a boring [Table of HMA Segment:Offset pairs](#) which shows how the number of pairs decreases to only one for the last 16 bytes of the HMA.



The highest byte in memory that can be accessed using the Segment:Offset scheme is:

FFFF:FFFF or  $FFFF0 + FFFF = 10\text{FFEF h.}$

## Normalized Segment:Offset Notation

Since there are so many different ways that a single byte in Memory might be referenced using Segment:Offset pairs, most programmers have agreed to use the same convention to *normalize* all such pairs into values that will always be **unique**. These unique pairs are called a **Normalized Address** or **Pointer**.

By confining the Offset to just the Hex values **0h** through **Fh** (16 hex digits); or a single **paragraph** and setting the Segment value accordingly, we have a unique way to reference all Segment:Offset Memory pair locations. To convert an arbitrary Segment:Offset pair into a normalized address or pointer is a two-step process that's



quite easy for an assembly programmer:

1. Convert the pairs into a single physical (linear) address.
2. Then simply insert the colon (:) between the last two hex digits!

For example, in order to *normalize* **1000:1B0F**, the steps are:

**1000:1B0F** → **11B0Fh** → **11B0:F** (or 11B0:000F)

Since the normalized form will always have three leading zero bytes in its Offset, programmers often write it with just the digit that counts as shown here: **11B0:F** (when you see an address like this, it's almost a sure sign that the author is using this *normalized* notation).

---


## How Segment:Offset notation can lead to Problems

The normalized notation for the first byte where all PC BIOS must place a floppy diskette's boot strap code is: **07C0:0** (or 07C0:0000). A big problem for some PC manufacturers came about when some BIOS writer *assumed* that there'd be nothing wrong with *jumping to* the bootstrap code at that particular Segment:Offset pair, *since* it's the same memory location as **0000:7C00**.

Somehow they never took into consideration the fact that the standard used by everyone else always set the SEGMENT values to ZERO. Therefore, a bootstrap code programmer could assume all Segment values (Code, Data, etc.) were zero and only have to deal with the Offset values in that Segment. Along comes this BIOS chip clone that sets the Code Segment to **07C0** (using a JMP 07C0:000 instruction), and suddenly there was a big problem getting most OS bootstrap code (including Microsoft® and IBM® OSs) to boot up in these computers! This is why some bootstrap code, such as that in the [GRUB](#) Boot Manager, will add extra instructions to make sure the Segment Registers have been set correctly! One of the authors of GRUB comments that his Long Jump code was necessary "because some *bogus* BIOSes jump to 07C0:0000 instead of 0000:7C00."

---




## Footnotes

<sup>1</sup>[\[Return to Text\]](#) KiB is the abbreviation for a *kibibyte* (a contraction of **kilo binary byte**) or *binary* kilobyte. It is equal to **2** to the **10th** power ( $2^{10}$ ) or **1024** bytes. Likewise, **MiB** is a *mebibyte* (**mega binary byte**); equal to **2** to the **20th** power ( $2^{20}$ ) or **1,048,576** bytes, and **GiB** is a *gibibyte* (**giga binary byte**); equal to **2** to the **30th** power ( $2^{30}$ ) or **1,073,741,824** bytes. In this document, which refers to memory in early IBM PCs (and the Intel 8086 and 80286 CPUs), we may at times refer to *kibibytes* using the abbreviation "**kb**" instead of KiB. (It is taking a long time for "kibi-", "mebi-" and "gibi-" to be recognized by techs and computer sales departments as the proper way to refer to memory ; even though they have been in official standards organizations for many years. See, for example,  [NIST: Prefixes for binary multiples](#), and

 [Adoption by IEC and NIST.](#))

**64 KiB** is also equal to **2** to the **16th** power [  $(2^{16}) = (2^{10}) \times (2^6) = (1024) \times (64) = 65,536$  ] bytes, and each of the two-byte (or 16-bit) registers in an 8086 CPU can *contain* a maximum value of: **1111 1111 1111 1111** in binary or **FFFFh** (hexadecimal). In decimal, that's: [  $(15 \times 16^3) + (15 \times 16^2) + (15 \times 16^1) + 15$  ] = [  $(15 \times 4096) + (15 \times 256) + (15 \times 16) + 15$  ] = 61,440 + 3,840 + 240 + 15 = 65,535. However, since memory always begins with **zero** (0) as its first location, that gives us  $65,535 + 1 = \mathbf{65,536}$  (or,  $16^4$ ) memory locations. **65,536** divided by 1024 per KiB = **64 KiB** of memory. For more on the use of Hexadecimal in computers, see: [What Is "Hexadecimal"?](#)

**2**[\[Return to Text\]](#) This is **4 gibibytes** (see Footnote #1) or **4** times  $(2^{30}) = 4,294,967,296$  bytes.

**3**[\[Return to Text\]](#) We've often heard that Bill Gates said something to the effect: '640K of memory should be enough for anyone.' Though many of us no longer believe he ever said those exact words (and he has finally made some public denials concerning this), he did, however, during a video interview with David Allison in 1993 for the National Museum of American History, Smithsonian Institution, say: "I laid out memory so the bottom 640K was general purpose RAM and the upper 384 I reserved for video and ROM, and things like that. That is why they talk about the 640K limit. It is actually a limit, not of the software, in any way, shape, or form, it is the limit of the microprocessor. That thing generates addresses, 20-bits addresses, that only can address a megabyte of memory. And, therefore, all the applications are tied to that limit. It was ten times what we had before. But to my surprise, we ran out of that address base for applications within -- oh five or six years people were complaining." (from  [a transcript of the interview](#), under the "Microsoft and the Mouse" section). For a bit more info, see:  [Did Bill Gates say the 640k line?](#) and perhaps of more interest to others, here are some  [verifiable quotes from Mr. Gates](#).

**4**[\[Return to Text\]](#) **1 MiB** of memory is **1,048,576** bytes ( $2^{20}$  bytes), but the Segment:Offset addressing scheme actually allows one to access up to **10FFEFh**, *plus one*, bytes of memory, or **1,114,096** bytes. We'll have more to say about this and the HMA (High Memory Area) shortly.

**5**[\[Return to Text\]](#) As we said above, until an IBM PC (or clone) actually had more than 1MiB of memory, it was expedient for the early IBM PCs to effectively *wrap-around* to the beginning of memory whenever programs tried to access an address past **FFFFh** bytes.  
[Sorry, this **FOOTNOTE IS STILL UNDER CONSTRUCTION!** It will soon have some links about the IBM PC AT model's keyboard and the infamous **A20** line!]

*Last Revised: 15 OCT 2007 (15.10.2007).*

---

You can write to me using this: [online reply form](#). (It opens in a new window.)

 [The Starman's ASSEMBLY Index Page](#)

 [The Starman's Realm Index Page](#)

