NOTAS ACERCA DE INTEL ASSEMBLY

<https://stackoverflow.com/questions/4903906/assembly-using-the-data-segment-register-ds>

# [Assembly: Using the Data Segment Register (DS)](https://stackoverflow.com/questions/4903906/assembly-using-the-data-segment-register-ds)

Currently I am in the midst of learning x86 assembly for fun, I'm love microcontroller programming, so I'm familiar with assembly.

Currently I've been searching high and low for the answer to this question, but can't seem to find it... the DS register, I know it's supposed to point to the global data in my program, but I don't know how it works exactly. I'm using NASM, and in most simple programs I see the following:

[org 0x7C00]

[bits 16]

main:

**mov ax, 0x0000**

**mov ds, ax**

mov al, [msg]

mov ah, 0x0E

mov bx, 0x0007

int 0x10

jmp $

msg db 'X'

times 510-($-$$) db 0

dw 0xAA55

and that works perfectly (even if I omit the bolded code), but how? Does the CPU automagically load the global variables starting at 0x0000? or is there something intrinsic here that I'm missing?

## **1 Answer**

When the computer is under real mode (the mode the CPU is at when the BIOS executes the bootloader), the method the CPU uses to calculate the address is very simple: Multiply segment register value by 16 (shift bits 4 positions to left), then add the offset.

For instance in an instruction like "mov ax, [0x1234]" the CPU would use "DS \* 0x10 + 0x1234" as the effective address (the first term resolves to zero in your case.) When you have one like "mov ax, [BP+0x32]" then the CPU will use "SS \* 0x10 + BP + 0x32". Note that now the CPU used a different segment register (the Stack Segment), and that is because when the BP register is used, the CPU assumes you wan't to access the stack by default (but you can override this by using [DS:BP + 0x32]).

More o less what I've explained and more can be found at <http://wiki.osdev.org/Real_Mode> and <http://www.internals.com/articles/protmode/realmode.htm> and lots of more places.

BTW, "msg" should be located more or less at 0x7C11 address.

<https://reverseengineering.stackexchange.com/questions/20135/how-is-ds-register-interpreted-in-x86-64-assembly-instructions>

# [How is DS register interpreted in x86-64 assembly instructions?](https://reverseengineering.stackexchange.com/questions/20135/how-is-ds-register-interpreted-in-x86-64-assembly-instructions)

IDA disassembled an x86-64 instruction as below:

0000000000000761 lea rdx, ds:0[rax\*4]

What I know is, segment registers are only relevant in in segmented memory model, while in case of paging they hold an index in GDT/LDT. Why is the above instruction, out of nowhere, referring to the ds segment register?

The fact that you are not using a segmented memory model doesn’t mean the instruction encoding doesn’t allow you to specify the segment :) Basically on x86\_64 the only relevant segment is gs, but a disassembler might still show other segment registers (implicitly) encoded in instructions...

– [mrexodia](https://reverseengineering.stackexchange.com/users/2221/mrexodia)

[Dec 18 '18 at 20:34](https://reverseengineering.stackexchange.com/questions/20135/how-is-ds-register-interpreted-in-x86-64-assembly-instructions#comment32541_20135)

## **1 Answer**

Segments are still used on 64 bit long mode and are still set up, except the CPU treats their bases to be 0 (except for gs and fs), and does not perform a limit check. The default segment for rax is indeed ds, but this can be changed with a segment override. lea rdx, ds:0[rax\*4] is lea rdx, ds:[rax\*4 + 0], which is lea rdx, ds:[rax\*4], which is the same as lea rdx, [rax\*4] in that when that instruction is executed, the AGU uses the ds segment descriptor by default for rax to perform privilege checks, but not limit or base checks. This segment descriptor is either renamed and is placed in the reservation station (and hence it is the job of the decoder and allocator to use the correct segment descriptor based on the prefix override or lack thereof) or is internal to the AGU and the AGU uses the uop opcode which might encode the segment to use. I have also seen lea rdx, [ds:rax] notation, which is the same thing as lea rdx, ds:[rax], but the problem is lea rdx, [ds:rax\*4] would be semantically misleading, as on x86 you can only do lea rdx, ds:[rax\*4]. I believe that gs:[0x32], gs:0x32 and [gs:0x32] all represent the same thing in x86 assemblies / disassemblies, for instance gs:0x32 does not mean 'get the calculated linear address of gs:0x32', because we have lea for that, and gs:[0x32] does not mean 'get the address at value at ds:[0x32] and then use that as an offset into gs and get the calculated linear address', therefore the other 2 forms can safely represent [gs:0x32], and as mentioned gs:[...] is more clear as it acts as precedence demarcation when a scale or index is involved

<https://www.tek-tips.com/viewthread.cfm?qid=717198>

[**syskplim**](https://www.tek-tips.com/userinfo.cfm?member=syskplim) (TechnicalUser)

**(OP)**

27 Nov 03 18:08

ES:0000 always point to the beginning and  
SS:SP always point to the end of the program in memory  
  
is this the truth?    
  
as i know, the sequence of the segments is DS,CS,SS,ES?  
  
.

---->ASM fresher need help!<----  
  
With kind regards.  
[**syskplim@streamyx.com**](mailto:syskplim@streamyx.com)

[**OSProgrammer**](https://www.tek-tips.com/userinfo.cfm?member=OSProgrammer) (Programmer)28 Nov 03 03:22

Hi syskplim!  
  
DS is called data segment register. It points to the segment of the data used by the running program. You can point this to anywhere you want as long as it contains the desired data. ES is called extra segment register. It is usually used with DI and doing pointers things. The couple DS:SI and ES:DI are commonly used to do string operations. SS is called stack segment register. It points to stack segment.  
  
The register SI and DI are called index registers. These registers are usually used to process arrays or strings. SI is called source index and DI is destination index. As the name follows, SI is always pointed to the source array and DI is always pointed to the destination. This is usually used to move a block of data, such as records (or structures) and arrays. These register is commonly coupled with DS and ES.  
  
The register BP, SP, and IP are called pointer registers. BP is base pointer, SP is stack pointer, and IP is instruction pointer. Usually BP is used for preserving space to use local variables. SP is used to point the current stack. Although SP can be modified easily, you must be cautious. It's because doing the wrong thing with this register could cause your program in ruin. IP denotes the current pointer of the running program. It is always coupled with CS and it is NOT modifiable. So, the couple of CS:IP is a pointer pointing to the current instruction of running program. You can NOT access CS nor IP directly.  
  
Bye  
OSProgrammer

[**lionelhill**](https://www.tek-tips.com/userinfo.cfm?member=lionelhill) (TechnicalUser)28 Nov 03 05:54

Basically agree but a few comments:  
(1) You can read cs directly like any other segment register. You don't need to set cs directly using the same instructions as you'd set other registers, because instructions such as ret, jsr, jmp are the equivalents of pop, push/mov, and mov, for cs and ip.  
(2) Where the segments are is up to your operating system. For instance in old .com applications for dos, the code segment and data segment were the same (cs=ds). If you don't like the current arrangement, you can change it. For instance, a small piece of embedded assembly in a larger high-level language program can be coded as though it were a .com program and bracketed by  
push ds  
push cs  
pop ds  
......  
pop ds  
to set up its own local data segment.  
(3) Although es is usually used with di, and ds with si, this is something brought over from the string handling op codes, and there is no particular reason why you shouldn't mix and match these segment registers with any index register according to your needs. The bx register is also frequently used in pointer arithmetic.  
(4) From 386 there are also spare segment registers, fs and gs. Unlike es, they need a segment prefix to the op code, so they are slightly slower. You don't need to worry about the prefix; your assembler will code it for you.  
(5) Remember that protected world and dos-world are totally different in their attitude to segment registers. In dos 16 bit world the segment registers hold a physical address shifted 4 bits. You can even use them to hold a number that isn't an address at all, and there won't be complaints! In protected mode the segment registers hold an index to a table entry, and if you try to set them to an invalid reference there will be loud complaints from the operating system.  
(6) Even in dos-16-bit world it's handy to be aware of scaled indexing (mov eax, ds:[si\*4] etc.). It works, and although it loses efficiency in 16bit world, being a 32bit instruction, it can be useful in 386+ systems.

[**syskplim**](https://www.tek-tips.com/userinfo.cfm?member=syskplim) (TechnicalUser)

**(OP)**

28 Nov 03 13:59

Since the stack-segment is always the last segment in an EXE-file, ES:0000 points to the beginning and SS:SP to the end of the program in memory. Through this the length of the program can be calculated  
and  
  
setfree proc near  
  
mov bx,ss ;first subtract the two segment addresses  
mov ax,es ;from each other. The result is  
sub bx,ax ;number of paragraphs from PSP  
          ;to the beginning of the stack  
mov ax,sp ;since the stackpointer is at the end of  
mov cl,4  ;the stack segment, its content indicates  
shr ax,cl ;the length of the stack  
add bx,ax ;add to current length  
inc bx    ;as precaution add another paragraph  
mov ah,4ah;pass new length to DOS  
int 21h  
ret       ;back to caller  
  
setfree endp  
  
  
  
as the code shown above, seems that ES is at higher location than SS, so SS-ES can get the length of paragraph.  
it this always the truth? ES always in lower address? how abt the other? CS and DS locations? (higher location will get lower address)  
  
i get stuck here!  :(  
  
  
.

---->ASM fresher need help!<----  
  
With kind regards.  
[**syskplim@streamyx.com**](mailto:syskplim@streamyx.com)

[**zeitghost**](https://www.tek-tips.com/userinfo.cfm?member=zeitghost) (Programmer)28 Nov 03 15:24

I think this indicates the initial state of the registers when the EXE program is loaded by the Dos Loader.  
  
According to Ray Duncan's Advanced Msdos Programming,  
  
at entry to an EXE program  
  
ds and es address the PSP segment  
  
ss addresses the stack segment, with sp addressing the top of stack.  
  
so sutracting es from ss will give the number of paragraphs in the PSP, code and data segments.  
  
Dividing sp by 16 gives the number of paragraphs in the stack segment.  
  
And adding the number of paragraphs in the stack segment to the number of paragraphs in the data, code, and PSP gives the length of the complete EXE program.  
  
Your code above also gives an extra one paragraph just to make sure that the top of stack doesn't get overwritten by something that gets loaded above this EXE file (since int 21h function 4ah resizes the memory block of this EXE).  
  
hope this helps  
rgds  
Zeit.

[**syskplim**](https://www.tek-tips.com/userinfo.cfm?member=syskplim) (TechnicalUser)

**(OP)**

6 Dec 03 16:16

Zeit,  
  
may i waste ur time to make me clear? https://www.tipmaster.com/images/smilewink.gif  
  
1. why?? and what is this in above code?  
  
mov cl,4  ;the stack segment, its content indicates  
shr ax,cl ;the length of the stack  
  
  
  
2. this is my debug code:  
  
file1.exe  
Ax=0000  Bx=0000  Cx=000a  Dx=0000  Sp=0400  Bp=Si=Di=0000  
Ds=0b22  Es=0b22  Ss=0b33  Cs=0b32  Ip=0000  Flag Reg  
  
file2.exe  
Ax=0000  Bx=0000  Cx=0012  Dx=0000  Sp=0400  Bp=Si=Di=0000  
Ds=0b22  Es=0b22  Ss=0b34  Cs=0b32  Ip=0000  Flag Reg  
  
file3.exe  
Ax=0000  Bx=0000  Cx=0038  Dx=0000  Sp=0400  Bp=Si=Di=0000  
Ds=0b22  Es=0b22  Ss=0b36  Cs=0b32  Ip=0000  Flag Reg  
  
  
can i find out the EXE size in memory after the program loaded?

---->ASM fresher need help!<----  
  
With kind regards.  
[**syskplim@streamyx.com**](mailto:syskplim@streamyx.com)

[**zeitghost**](https://www.tek-tips.com/userinfo.cfm?member=zeitghost) (Programmer)10 Dec 03 04:23

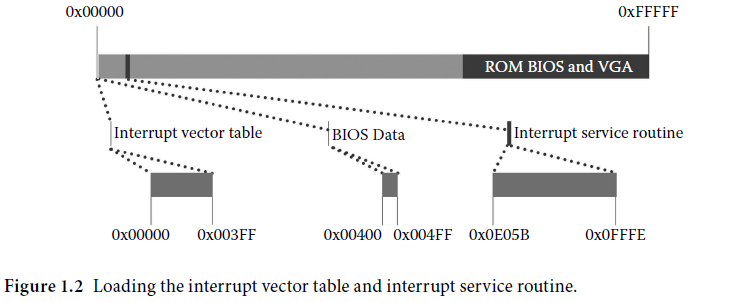
So far as I understand it, a paragraph is 16 bytes.  
  
The reason for defining the length of the program in paragraphs is that Int 21h funtion 4ah, resize memory block, requires the new size of the memory block in paragraphs, contained in the BX register.  
  
What Ray Duncan is saying is relevant when the program is initially loaded into memory, and before instructions to change the segment registers have executed.  
  
Your subroutine above is obviously executed after this.  
  
rgds  
Zeit.

[**zeitghost**](https://www.tek-tips.com/userinfo.cfm?member=zeitghost) (Programmer)21 Dec 03 10:59

Further to the above.  
  
On entry, from the dos loader, ES and DS point to the PSP.  
  
The order of DS and CS depends on how MASM is instructed to treat them.  
  
If the file contains  
  
DATA SEGMENT  
  
DATA ENDS  
  
CODE  SEGMENT  
  
CODE ENDS  
  
will generate the segments in that order unless told otherwise.  
  
This is not the recommended DOS order of segments, which is  
  
CODE SEGMENT  
  
CODE ENDS  
  
DATA SEGMENT  
  
DATA ENDS  
  
STACK SEGMENT  
  
STACK ENDS  
  
In masm 6, there's a directive DOSSEG which forces this.  
  
The linker can also force this using /DOSSEG as a command line parameter.  
  
Actually you can probably write a book about this.  
  
So I'll stop here.  
  
rgds  
Zeit.

REF:

[Lixiang\_Yang]\_The\_Art\_of\_Linux\_Kernel\_Design\_Ill(BookZZ.org).pdf, página 4 (19/524)



The BIOS puts the interrupt vector table at the beginning of the memory, which is

1 KB (0x00000–0x003FF).

The BIOS data area is next to it, 256 B (0x00400–0x004FF),

and then the interrupt service routine (8 KB), 56 KB, comes after it (0x0E05B). Figure 1.2

shows the exact locations.

**Tip:**

Note that 0x00100 is 256 bytes and 0x00400 is 4 × 256 bytes = 1024 bytes, or

1 KB. Since it is from 0x00000, the high section of the 1 KB is not 0x00400 but

0x00400-1 instead, which is 0x003FF.

The interrupt vector table has 256 interrupt vectors and 4 bytes for each vector, including

2 bytes for CS and 2 bytes for IP. Each interrupt vector points to a particular interrupt

service routine.

We will explain in detail how to use these interrupt service routines to load OS kernel

into the memory.

**Tip:**

INT: interrupt. As its name suggests, INT refers to an interrupt of an ongoing

process. An external event interrupts the program that is being executed, to run

a specific procedure to handle this event. After the INT procedure is done, the

interrupted program will continue. Interrupts are quite similar to the function

call in C.

Interrupt means a lot to the OS; we will discuss it further later on.

Page 9 (24/524)

[real mode]

Comment

jmpi go, INITSEG

go: mov ax,cs

These two lines of codes are very tricky. After Bootsect copies itself, the contents in

0x07C00 and 0x90000 are the same. Please note that before “*jmpi go, INITSEG*,” CS is

0x07C0. After that, CS becomes 0x9000. Then, it executes the next line, “*mov ax,cs.*” It

is a good way to “jump and continue performing the same codes.”