Data definition directives

Your CODE segment starts at offset 00401000 (or 00402000) in OllyDbg
Always your DATA segment starts at offset 00402000 (or 00401000) in OllyDbg

Segment data

a1 db 0,1,2,'xyz' ; 00 01 02 'x' 'y' 'z' ; offset(a1) – computed at OllyDbg loading time = 00402000; offset(a1) determined at assembly time by NASM = 0 !!!
78 79 7A
db 300, "F"+3 ; 2C ascii code for 'F' + 3 = 49 ; Warning – byte data (300) exceeds bounds!

a2 TIMES 3 db 44h ; 44 44 44 ; offset(a2) = 00402008, assembly time offset = 8 !

a3 TIMES 11 db 5,1,3 ; 05 01 03…11 times (total 33 bytes)

a41 db a2+1 – syntax err. – OBJ format can only handle 16 or 32 bits relocation !

a4 dw a2+1, 'bc' ; offset(a2)=00402008h; a2+1=…2009h; 09 20 'b' 'c' = 09 20 62 63
; 09 20 (correct, BUT… this particular 20h value it is only finally computable after LOADING !!!) – so the offset of the beginning of segments is computable only at LOADING TIME !!!)
The variables offset relative to the beginning of segments in which they are defined are constant POINTERS (so POINTER data types !, NOT scalar !) determinable at assembly time !!

a42 dw a2+1, 'bc' ; offset(a2)=00402008h; a2+1=2009h; 09 10 'b' 'c' = 09 20 62 63 !
'b', 'c' ; 'b' 00 'c' 00 = 09 20 62 00 63 00

a44 dw 2009h ; 09 20

a5 dd a2+1, 'bcd' ; 09 20 40 00| 62 63 64 00
a6 TIMES 4 db '13' ; equiv with a6 TIMES 4 db '1','3' ; 31 33 31 33 31 33 31 33
a6bis TIMES 4 dw '13' ; 31 33 31 33 31 33 31 33 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

a7 db a2 ; syntax err. – OBJ format can only handle 16 or 32 bits relocation (equiv. mov ah,a2)
a8 dw a2 ; 08 20
a9 dd a2 ; 08 20 40 00
a10 dq a2 ; 08 20 40 00 00 00 00 00

a11 db [a2] ; syntax error ! A CONTENTS of a memory area or the contents of a register are NOT values accessible or determinable at assembly time ! These are accessible and determinable ONLY at run-time !!

```
a12 dw [a2] ; syntax error !

a13 dd dword [a2] ; syntax error !
a14 dq [a2] ; syntax error !

a15 dd eax; expression syntax error
a16 dd [eax]; expression syntax error
```

```
mov ax, v  ;  Warning – 32 bit offset in 16 bit field !!!
```

Segment code (starts at offset 00401000 – WHO DECIDES THAT ?)

The linker makes such decisions. The base address for loading PEs, at least the default one (set by the Microsoft linkeditor and not only) is 0x400000 for executables (respectively 0x10000000 for libraries). Alink complies with this convention and fills in the ImageBase field of the IMAGE_OPTIONAL_HEADER structure in the P.E. newly built value 0x400000. As each “segment” / section of the program can provide different access rights (the code is executable, we can have read-only segments etc ...), these are planned to start each one at the address of a new memory page (4KiB, so multiple of 0x1000), each memory page can be configured with specific rights by the program loader. In the case of small programs, the implication is that you will get the following map of the program in memory (at run time):

- the program is planned to be loaded into memory at exactly the address 0x400000 (but here will reach the metadata structures of the file, not the code or the data of the program itself)

- the first "segment" will be loaded at 0x401000 (quotation marks because it is not a segment itself but only a logical division of the program, the "segment" of a segment register is not directly associated - for this reason the name is often preferred section instead of segment)

- the second "segment" will be loaded at 0x402000 (the segment that the processor will use for segmentation starts at address 0 and has a limit of 4GiB, regardless of the addresses and section sizes)

- will be prepared "segment" (section) of imports, "segment" of exports and "segment" of stack in the order decided by the likeditor (and of dimensions also provided by him), segments that will be loaded from 0x403000, 0x404000 and so on (increments of 0x1000 as long as they are small enough, otherwise need to be used for increment the smallest multiple of 0x1000 which allows enough space for the contents of the entire segment)

According to the decision logic of the start addresses of the sections, we can conclude that here we have a section (probably data) before the code, containing less than 0x1000 bytes, which is why the code starts immediately after, from 0x402000, the program map being at the end : metadata (headers) from 0x400000, data to 0x401000 and code to 0x402000 (followed of course by other "segments" for stack, imports and, optionally, exports).

Linkeditorul ia deciziile de acest tip. Adresa de baza pentru incarcarea PE-urilor, cel putin cea implicita (setata de catre linkeditorul de la Microsoft si nu numai) este 0x400000 in cazul executabilelor (respectiv 0x10000000 pentru biblioteci). Alink respecta aceasta conventie si completeaza in campul ImageBase al structurii IMAGE_OPTIONAL_HEADER din fisierul P.E. nou construit valoarea 0x400000. Cum fiecare "segment"/sectiune din program poate prevedea drepturi diferite de acces (codul este executabil, putem avea segmente read-only etc...), acestea sunt planificate sa inceapa fiecare la adresa cate unei noi pagini de memorie (4KiB, deci multiplu de 0x1000), fiecare pagina de memorie putand fi configurata cu drepturi specifice de catre incarcatorul de programe. In cazul unor programe de mici dimensiuni, implicatia este ca se va obtine urmatoarea harta a programului in memorie (la executare):

- programul este planificat a fi incarcat in memorie la exact adresa 0x400000 (insa aici vor ajunge structurile de metadate ale fisierului, nu codul sau datele programului in sine)
- primul "segment" va fi incarcat la 0x401000 (pun ghilimele deoarece nu este un segment propriu-zis ci doar o diviziune logica a programului, nu este asociat direct "segmentul" unui registru de segment – din aceasta pricina se prefera de multe ori denumirea de sectiune in loc de cea de segment)
- al doilea "segment" va fi incarcat la 0x402000 (segmentul pe care il va folosi procesorul pentru segmentare incepe la adresa 0 si are limita de 4GiB, indiferent de adresele si dimensiunile sectiunilor)
- va fi pregatit "segment" (sectiune) de importuri, "segment" de exporturi si "segment" de stack in ordinea decisa de catre likeditor (si de dimensiuni prevazute tot de catre acesta), segmente ce vor fi incarcate de la 0x403000, 0x404000 si asa mai departe (incremente de 0x1000 cat timp au dimensiune suficient de mica, in caz contrar fiind nevoie a se folosi pentru increment cel mai mic multiplu de 0x1000 care permite suficient spatiu pentru continutul intregului segment)

Conform logicii de decizie a adreselor de inceput ale sectiunilor, putem concluziona ca aici avem o sectiune (de date probabil) inaintea celei de cod, continand sub 0x1000 octeti, motiv pentru care codul porneste imediat dupa, de la 0x402000, harta programului fiind la final: metadate (antete) de la 0x400000, date la 0x401000 si cod la 0x402000 (urmat bineinteles de alte "segmente" pentru stiva, importuri si, optional, exporturi).

Segment code (starts at offset 00401000)

```
Start:
        Jmp Real_start    (2 bytes)   - 00401000
        a db 17                       - 00401002
        b dw 1234h                    - 00401003
        c dd  12345678h               - 00401005

Real_start:
        …….
        Mov eax, c ; eax = 401005
```

Mov edx, [c] ; mov edx, DWORD PTR DS:[401005]; in mod normal asta inseamna ca in EDX will be assigned with the doubleword of offset 00401005 taken from DS !!!!

……
Mov edx, [CS:c] ; mov edx, DWORD PTR CS:[401005]
Mov edx, [DS:c] ; mov edx, DWORD PTR DS:[401005]
Mov edx, [SS:c] ; mov edx, DWORD PTR SS:[401005]
Mov edx, [ES:c] ; mov edx, DWORD PTR ES:[401005]

The output will be in all of the 5 cases the same EDX:=12345678h WHY ??

The explanation is directly related to the flat memory model - all segments actually describe the entire memory, from 0 to the end of the first 4GiB of memory. As such, [CS: c] or [DS: c] or [SS: c] or [ES: c] will access the same memory location but with different access rights. Although all selectors indicate identical segments in address and size, they may differ in how other control and access fields of the segment descriptors indicated by them are completed.

The flat model assures us that the segmentation mechanism is transparent to us, we do not notice differences between segments and, as such, we completely get rid of the segmentation worry, but we are interested in the logical division into segments of the program, which is why we use separate sections / "segments". for data code). This is true but only as long as we limit ourselves to CS / DS / ES and SS! The FS and GS selectors point to special segments that do not follow the flat pattern (reserved for the interaction of the program with the S.O.), more precisely, [FS: c] does not indicate the same memory as [CS: c]!

Explicatia este direct legata de modelul de memorie flat – toate segmentele descriu in realitate intreaga memorie, incepand de la 0 si pana la capatul primilor 4GiB ai memoriei. Ca atare, [CS:c] sau [DS:c] sau [SS:c] sau [ES:c] vor accesa aceeasi locatie de memorie insa cu drepturi de acces potential diferite. Desi toti selectorii indica segmente identice ca adresa si dimensiune, acestia pot avea diferente in cum le sunt completate alte campuri de control si de acces ale descriptorilor de segment indicati de catre ei.

Modelul flat ne asigura ca mecanismul de segmentare este transparent pentru noi, noi nu sesizam diferente intre segmente si, ca atare, scapam complet de grija segmentarii (insa ne intereseaza impartirea logica in segmente a programului, motiv pentru care folosim sectiuni/”segmente” separate pentru cod date). Acest lucru este valabil insa doar cat timp ne limitam la CS/DS/ES si SS! Selectorii FS si GS indica inspre segmente speciale care nu respecta modelul flat (rezervate interactiunii programului cu S.O-ul), mai precis, [FS:c] sau [GS:c] NU indica aceeasi memorie ca si [CS:c]!