The **A**rt **O**f **D**isassembly

(C) The Reverse-Engineering-Network

18th November 2003
- ALPHA Release -

Contents

Chapter 1

Preface

Welcome to "The Art Of Disassembly" - a free online ebook for writing a disassembler in pure assembly. As you may notice the book is not finished yet. The state of the ebook is ALPHA. So please be patient with us. We have included now 14 chapters which provide you the basic knowledge you need for understanding how a disassembler works. We work hard to finish this ebook but you have to give us time - so please do not contact us with questions when everything is finished.

As well we have added for each chapter a small source package which you can use for playing with the code. This handbook was mainly written by CuteDevil and checked for errors by Pegasus[REN] and Ben.

The book will be always for free and there is no intention to make it ever commercial. If you have any comments or suggestions feel free to contact us over the reverse-code-engineering community board at http://board.anticrack.de. For the latest release please refer to http://aod.anticrack.de.

Why are we doing this project ? I don´t know. Knowledge increasing, fun, time-wasting... Why are you interested in writing a disassembler ?

Zero - Publisher (Reverse-Engineering-Networks, REN)
CuteDevil - Main Author
Pegasus - Chief Corrector (Reverse-Engineering-Networks, REN)

Chapter 2

Disassembler Skeleton

2.1 What do we need in a disassembler?

If we take a look at any GUI disassembler (IDA, W32DASM, BDASM,...) we will quickly notice the main parts of the disassembler structure:

- MENU:
  To access the main features (File handling, Viewing, Searching,...)
- LISTVIEW:
  the main view, to display (Opcodes, Mnemonics, Comments,...)

There are other subparts that we need:

- TOOLBAR:
  To ease the access of the main features
- LISTVIEW:
  To display the important events (Opening, Checking, Disassembling, other info about the target file)

2.2 What tools do we need?

As mentioned before, we'll use RadASM [3] (With MASM 7 [1]) as our IDE & programming language.

2.3 Constructing a simple skeleton

Okay, let's do it! Here's what we'll do..

1. Selecting a "'New Project"' from RadASM opens the Project Wizard for us
2. Accept the default settings for (MASM as the assembler, Win32 App as the project type)
3. For the project name we choose (Skeleton)
4. For the project description we choose (Simple Skeleton)
5. Choose the destination folder for the project
6. For the template we choose (Dialog application –> DialogApp.tpl)
7. We click 'Next' and accept the default options

Now all the files needed for the project are included, so we compile the project and run it.

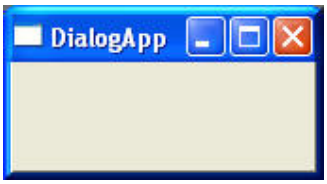


Figure 2.1: The empty program after compiling and running resized.

Now, in the 'Resources' section, we open the 'Skeleton.dlg' to make the necessary changes.

1. Change the size to a convenient one (600 X 600) for now
2. Change the name to IDD_MAIN
3. Disable the (MaxButton) for now
4. Disable the (SizeBorders) for now
5. Compile and run

Okay, the main dialog is now suitable, we add the needed resources for the disassembler , so in order to do so, we have to step through the following points:

1. To add the menu, we add a new (Main.mnu) to the project, the 'Menu Editor' appears, we construct it so we will get a menu-entry "'(File-> Open/Close/Exit)"':

```
#define IDR_MENU 10000
#define IDM_FILE 10001
#define IDM_OPEN 10002
#define IDM_CLOSE 10003
#define IDM_EXIT 10004
IDR_MENU MENU
BEGIN
POPUP "&File"
BEGIN
MENUITEM "Open\tCtrl+O",IDM_OPEN
MENUITEM "Close",IDM_CLOSE
MENUITEM "Exit\tAlt+X",IDM_EXIT
END
END
```

2. From the 'IDD_MAIN Properties' choose our 'Main.mnu' as the default MENU
3. We add a 'LISTVIEW' control to be the main disassembler view
4. We add at the bottom a simple LISTBOX to display the important events as they happen
5. We compile and run

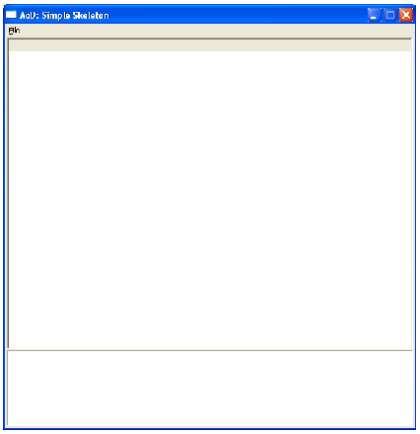


Figure 2.2: The empty program after compiling and running resized - Part II.

Although it doesn't look nice, we've made the first part. Next we'll see how can we (beautify) its look.

2.4 Some changes in the code

2.4.1 In (Skeleton.inc)

```
.const
IDC_DASM    equ      1001
```

and

```
.data?
hDsmList   dd         ?    Handle of the main ListView
hWnd       dd         ?    Handle of the main dialog
```

With *IDC_DASM* as constant for the main ListView. We need this constant when we retrieves the handle of the ListView using *GetDlgItem()*. *hDsmList* golds the handle of the main ListView because we need its value as we'll see. *hWnd* Holds the handle of the main dialog, we need this one too.

2.4.2 In (Skeleton.asm)

```
.code
PrepareListView proc
;-------------------------------------------------------------------------;
;        Prepare the main ListView and changes its color & style          ;
;-------------------------------------------------------------------------;
Invoke GetDlgItem,hWnd,IDC_DASM         ; Gets its hWnd
mov hDsmList, eax
invoke SendMessage, hDsmList, LVM_SETTEXTCOLOR, 0, 0FE7000h ; Sets its color
invoke SendMessage, hDsmList, LVM_SETEXTENDEDLISTVIEWSTYLE, 0,\ ; Sets its style
LVS_EX_FULLROWSELECT or \
LVS_EX_GRIDLINES + LVS_EX_FLATSB
ret
PrepareListView endp
```

This procedure mainly, sets the style needed for our disassembler, and sets the background color too. We will not notice the difference before we insert some text into the ListView, but this is another story... And another chapter...

Notice:
We also add this code, to invoke our code at the start of the program (initialization):

```
.if eax==WM_INITDIALOG
    invoke PrepareListView ; Prepare our ListView
.elseif eax==WM_COMMAND
```

2.5 Final Words

In this chapter we have not actually started. We have just prepared ourselves and have become ready to go on our long journey in the disassembler project.

2.6 In the next chapter

We will:

- Write the menu handling code
- Know how to (Open/Map/Access/Close) Files
- Learn with example how does LISTVIEW work in assembly
- Much more!

Chapter 3

Bringing Life to the Skeleton

3.1 Handling our menu code

In the previous chapter we designed the skeleton, but we did not write the code to handle anything. Now, we are going to write the menu handling part.

We designed our menu to consist of[1]:

- *Open:* To open the file we want to disassemble. Later we will invoke the disassembling engine.
- *Close:* To close it.
- *Exit:* To exit the program..

The *Open* part will need the most of our work as it contains file handling and other stuff as we will see.

3.1.1 Handling our DlgProc code

We know that when we press a menu item, it sends the message *WM_ COMMAND* to our program. And we need to handle this message. We also know that each item has its own handle, which we can get using *wParam*. So first we add the menu items constants to our *Skeleton.inc*:

```
IDM_OPEN   equ  10002
IDM_CLOSE  equ  10003
IDM_EXIT   equ  10004
```

And in the (Skeleton.asm) in the DlgProc Section:

```
.elseif eax==WM_COMMAND
   mov eax, wParam
   .if ax==IDM_OPEN
      ; We put our code here
   .elseif ax==IDM_CLOSE
      ; .......
   .elseif ax==IDM_EXIT
      ; .......
   .endif
.elseif eax==WM_CLOSE
```

Now we handled the code for our three menu items. We have time now to write each ones code.

[1] More options will be added later.

3.1.2 Open menu item

So, what do we really need to do here?

- A dialog box to allow user to choose the file
- Opening the file and Mapping it to memory

For displaying the the (*Open File*) dialog box we need to add some data to (*Skeleton.inc*) first[2]:

```
ofn   OPENFILENAME <>
buffer       db  260  dup(0)
StrFilter    db "Executable Files",0,"*.exe",0
             db "Dynamic Link Libraries",0,"*.dll",0,0
```

You should now have a look at the supplied source code with this book.

Then, in order to use our *OPENFILENAME* structure we need to initialize it first. In the *DlgProc* we need to add this code:

```
.if eax==WM_INITDIALOG
;------------------------------------ Initializing our dialog --------

;=========================================
;       Prepare our ListView
;=========================================

invoke PrepareListView

;=========================================
;   Initialize the OPENFILENAME structure
;=========================================

mov ofn.lStructSize, SIZEOF ofn
push hWnd
pop ofn.hwndOwner
         push hInstance
         pop  ofn.hInstance
         mov  ofn.lpstrFilter, OFFSET StrFilter  ; ( *.exe & *.dll )
         mov  ofn.lpstrFile, OFFSET buffer  ; ( Store the file name )
         mov  ofn.nMaxFile, 260

; ( File must exists/ Hide read only files)
         mov  ofn.Flags, OFN_FILEMUSTEXIST or \
           OFN_PATHMUSTEXIST or OFN_LONGNAMES or\
                  OFN_EXPLORER or OFN_HIDEREADONLY
```

This code only initializes our structure so we put it in the *WM_INITDIALOG* section. For popup the dialog box and to display it we need to invoke the function *GetOpenFileName*.

[2]For *OPENFILENAME* please refer to MSDN [6] for more details.

So back to *WM_ COMMAND* we add this code:

```
.elseif eax==WM_COMMAND
   mov eax, wParam
   .if ax==IDM_OPEN
   ;------------------------------------ Open file -------------------
   ; Display the "Open File" dialog box
   invoke GetOpenFileName, ADDR ofn
 ;-----------------------------------------------------------------------
```

And finally we have a working OpenFile dialog box !

Figure 3.1: Bringing Life to the Skeleton.

After opening the dialog box we need to do a simple check to see if we really chose a file or if we clicked cancel or something similar.

Right after the: `invoke GetOpenFileName, ADDR ofn`

We continue like this:

```
.if eax==TRUE
   invoke CreateFile,ADDR buffer,\
      GENERIC_READ or GENERIC_WRITE ,\
      FILE_SHARE_READ or FILE_SHARE_WRITE,\
      NULL,OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,\
      NULL
   mov hFile,eax  ; Stores the handle of the file
.endif
```

Notice that we need to add this to the (*Skeleton.inc*):

```
 hFile  dd  ?
```

The function *CreateFile* returns the handle of the file and then we store it in (*hFile*). After this we need to map the file to memory for the ease of access using the function *CreateFileMapping* to create a file mapping object then use the function *MapViewOfFile* to map the file to memory. But to use this last function we need to get the size of the file we need to map so we will use *GetFileSize* to do the work for us.

So in short[3] we do:

- Get file handle using CreateFile
- Create file mapping object using CreateFileMapping
- Get the file size using GetFileSize
- Map the file to memory using MapViewOfFile

We also check - after each function - for errors. If one happened then we should display a message box. To declare these error messages we create a new file called (*Msgs.inc*) containing all the error messages we need. Currently we need only two messages.

Please check the sources now, or you will get lost!

After this we almost finished the code needed for the open file section. But as you might have noticed the code is - though not much at all - long and for the simplicity we need in our code we will move this code (*File Open*) into an independent procedure. Let us call it (*MnuFileOpen*). We define it as:

```
MnuFileOpen proc
   ......
   ......
   ret
MnuFileOpen endp
```

> **Hint:** *One nice feature of RadASM [3] is that you can (Collapse/Expand) any procedure you want, so the code looks simpler!*

And simply move our code for the (*File opening*) to this new procedure and change the code to be like this:

```
.if    ax==IDM_OPEN
   invoke MnuFileOpen
.elseif ax==IDM_CLOSE
```

Now we can continue. We have finished the *Open* part - for this chapter - so now let us continue to the *Close* part.

[3]Please check source code for all the details

3.1.3 Close menu item

What do we really need to do in this part? Remember that when we opened the file and mapped it to memory it simply uses the memory. So we need to free this memory before closing. We use *UnmapViewOfFile* to fully close the mapped object and we must unmap all the mapped view. Additionally we use *CloseHandle* to close the file mapping object. We need to write the code to free the memory too.

So as we did above we will make an independent procedure called MnuFileClose and call it like this:

```
.elseif ax==IDM_CLOSE
   invoke MnuFileClose
.elseif ax==IDM_EXIT
```

So what is actually in our *MnuFileClose* ? It is like this:

```
MnuFileClose proc
   cmp hFileMapping, 0
   jnz @f
   ret
   @@:
   invoke UnmapViewOfFile, FileOffset
   invoke CloseHandle, hFile
   mov FileOffset, 0
   mov hFile, 0
   ret
MnuFileClose endp
```

What this procedure does is checking if there is a file mapped to memory[4] and if so then unmap it and close its handle.

Finally - for this chapter - we have to code the Exit part.

3.1.4 Exit menu item

The easiest one, we need to simply close the file (if still opened) and then exit the program.

```
.elseif ax==IDM_EXIT
   invoke MnuFileClose
   invoke ExitProcess,0
.endif
```

This simply invokes the *MnuFileClose* procedure to make sure the file is closed and unmapped before we close the program, then calls *ExitProcess* to exit the program.

[4] check it´s handle

The menu handling code is finished for this chapter. One more thing you should notice in the source code:

- *GetMenu:* at the initialization of the dialog
- *EnableMenuItem:* at the (*MnuFileClose*/*MnuFileOpen*)

I think it does not need much to figure out what it is doing.

Included is a project file (*MsgBox*) for a small program. You can test it with our disassembler.

3.2 Final Words

In this chapter we wrote the code that handles the menu and saw how to open the file and map it to memory. It was supposed that we will see how *ListView* works, but I have decided to make it in an independent chapter so it can be explained in details with examples.

3.3 In the next chapter

We will:

- Understand with example how *ListView* does work

Chapter 4

The ListView

4.1 What is a ListView ?

A *ListView* is one of the windows common controls like *RichEdit*, *ProgressBar*, *TreeView* and many others. In someway it is like the *Listbox* but with more enhanced capabilities. *ListView* has four methods for viewing data - *Icon, Small Icon, List, Report* - but we will only focus on the last method.

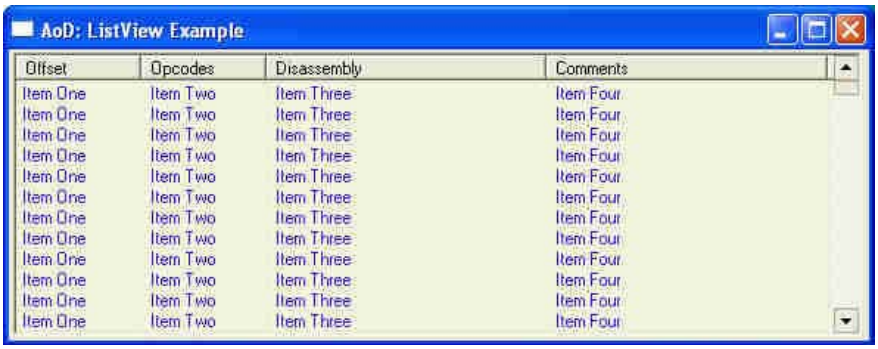


Figure 4.1: Example of a ListView.

4.2 Using the ListView control

In RadASM [3] we do this:

1. New Project: with project name (*ListView*) and template (*DialogApp.tpl*)
2. We change the caption to "'ListView Example"'
3. In the (*ListView.dlg*) we draw a ListView control, and name it (*IDC_DSM*)
4. Choose Report as the ListView type

Compile the program and run it. Now we obviously have a nice empty *ListView*, so how to fill it?

In the *ListView* (Report View) there are one or more columns and the arrangement of data can be thought of as a table arranged in columns and rows. But how can we add columns? We can do this by sending *LVM_INSERTCOLUMN* message to our ListView control. So we need to know how to send this message explicitly.

```
LVM_INSERTCOLUMN
wParam = iCol
lParam = pointer to a LV_COLUMN structure
```

And what is *LV_COLUMN* structure? Obviously it is a structure which contains the information about the column we need to insert to our *ListView*.

```
LV_COLUMN STRUCT
   imask            dd     ?
   fmt              dd     ?
   lx               dd     ?
   pszText          dd     ?
   cchTextMax       dd     ?
   iSubItem         dd     ?
   iImage           dd     ?
   iOrder           dd     ?
LV_COLUMN ENDS
```

Seems like we need some more details.

imask contains some flags that specify which members of the structure are valid, because some members are not used every time, but only in some situations.

- *LVCF_FMT:* It means that (fmt) member is valid
- *LVCF_SUBITEM:* The iSubItem member is valid
- *LVCF_TEXT:* The pszText member is valid
- *LVCF_WIDTH:* The lx member is valid

fmt specifies the alignment of the item or subitem in the column. It can be one of the following values

- *LVCFMT_CENTER:* Text is centered
- *LVCFMT_RIGHT:* Text is right-aligned
- *LVCFMT_LEFT:* Text is left-aligned

lx contains The width of the column (in pixels) (The width of the column can be changed later in runtime using *LVM_SETCOLUMNWIDTH* message)

pszText is a pointer to the name of the column we want to insert. If the message is sent to get properties of the column, this item contains a pointer to a large buffer and the field cchTextMax must contain the size of the buffer.

cchTextMax contains the size (in bytes) of the buffer in the item (pszText) above. This item is only used when we're getting the properties of the column.

- *iSubItem:* index of the subitem that's associated with the column.
- *iImage:* Index (zero-based) for an image in an image list. (not important to us)
- *iOrder:* Zero-based column offset in a left-to-right order. (not important to us)

So basically after any *ListView* is created we should insert one or more columns into it. We must do this, because we'll use the report view for our *ListView*. In order to do this, we need to construct a valid *LV_COLUMN* structure, fill it with the necessary information and then send the structure to our *ListView* using *LVM_INSERTCOLUMN* message.

As we will add more than one column, and we really need to make our program as simple as we can, we will make a new procedure for adding columns to the *ListView*:

```
InsertColumn proc pszHeading:DWORD, dwWidth:DWORD
   ;=========================================================
   ; pszHeading: Pointer to the name of the column
   ; dwWidth: Width of the column we want to insert
   ;=========================================================
   LOCAL lvc:LV_COLUMN
   mov lvc.imask, LVCF_TEXT + LVCF_WIDTH
   push pszHeading
   pop lvc.pszText
   push dwWidth
   pop lvc.lx
   invoke SendMessage,hDsm, LVM_INSERTCOLUMN,dwIndex,addr lvc
   inc dwIndex
   ret
InsertColumn endp
```

What this procedure does, is simply constructing a *LV_COLUMN* structure, and fill the necessary fields (Text & Width) and then send the message *LVM_INSERTCOLUMN* to insert the column. This procedure takes two parameters, one is a pointer to the name of the column, the second is the width of the column we want to insert.

> **Notice:** The *dwIndex* is a counter to how many columns are there, so the new columns is inserted on the right of the old column.
>
> **Notice:** The message is sent to the *ListView* with the handle (*hDsm*) so, we must get the handle of our *ListView* at the initialization time of the dialog, using *GetDlgItem* and then store the handle to *hDsm*.

We can now insert any column we want to use:

```
invoke InsertColumn,addr TextOfTheColumn,WidthOfTheColumn
```

Please check the sources now, or you will get lost!

We now know how to add columns - but how to add data to each column? First, we must know something about *ListView*. Items are the main entries in a *ListView*. In the *report view*, there are items and subitems. Items are the items in the leftmost column, while subitems are the items in the remaining columns.

```
Column1   Column2   Column3   Column4   Column5
Item1     Subitem1  Subitem2  Subitem3  Subitem4
Item2     Subitem1  Subitem2  Subitem3  Subitem4
Item3     Subitem1  Subitem2  Subitem3  Subitem4
```

So how can we add items? For adding columns we need to fill a structure (*LV_ITEM*) and then send it using *LVM_INSERTITEM* message.

The *LV_ITEM* structure is defined as:

```
LV_ITEM STRUCT
   imask         dd ?
   iItem         dd ?
   iSubItem      dd ?
   state         dd ?
   stateMask     dd ?
   pszText       dd ?
   cchTextMax    dd ?
   iImage        dd ?
   lParam        dd ?
   iIndent       dd ?
LV_ITEM ENDS
```

More details:

- *imask:* Some flags that specify which members of the structure are valid, similar to the imask above.
- *iItem:* Index (Zero-Based) for the item the structure is referring to. (Row number)
- *iSubItem:* Index (Zero-Based) for the subitem associated with the item specified by (iItem) above, it can be thought of as the field that contains the column.
- *state:* Some flags that tell the status of the item (selected/highlighted/focused/..). It can also contain an index (One-Based) for the overlay image or the state of the image used by the item.
- *stateMask:* As we said state member can contain the state flag or the overlay image index, we need to specify what value we're interested in.
- *pszText:* As in *LV_COLUMN* structure.
- *cchTextMax:* As in *LV_COLUMN* structure
- *iImage:* Index into an imagelist containing the icon for the *ListView* control.
- *lParam:* A value used by the user to specify how to sort items in the *ListView*.
- *iIndent:* We have nothing to do with it! Please see MSDN [6] for more details!

Notice: To add an Item we use *LVM_INSERTITEM* but to add a Subitem we use *LVM_SETITEM* and this is because subitems are considered properties of items. It means, you can not have a subitem without an item that is associated with it.

```
InsertItem proc uses ecx Row:DWORD, Column:DWORD, pszCaption:DWORD
   ;=============================================================
   ; Row: Zero-Based Index (Row no. for the item)
   ; Column: Zero-Based Index (Column no. for the item)
   ; pszCaption: Pointer to the name of the item to insert
   ;=============================================================
   LOCAL lvc:LV_ITEM
   mov lvc.imask, LVCF_TEXT
   push row
   pop lvc.iItem
   push Column
   pop lvc.pszText
   .if Column==0
      invoke SendMessage,hDsm, LVM_INSERTITEM,0,addr lvc
   .elseif
      invoke SendMessage,hDsm, LVM_SETITEM,0,addr lvc
   .endif
   inc dwIndex
   ret
InsertItem endp
```

What this procedure does is simply constructing a *LV_ITEM* structure and fill the necessary fields (Row & Column & Text) and then send the message *LVM_INSERTITEM* to insert an item. If the (*Column*) field is not 0 it means we we need to insert a subitem so we send the message *LVM_SETITEM* instead. This procedure takes three parameters, one is the row number (Zero-Based) the second is the column number, the third n is a pointer to the text of the item we wanna insert. We can now insert any item/subitem we want using:

```
   invoke InsertItem,Row,Column,addr TextOfTheItem
```

Please check the sources now, or you will get lost!

After we learned how *ListView* work, and how to add column, items and subitems, let's do a quick change to the look of our *ListView*[1]. Let us do this:

```
   invoke SendMessage, hDsm, LVM_SETTEXTCOLOR, 0, 00E41030h
   invoke SendMessage, hDsm, LVM_SETBKCOLOR,0,00DEF5F3h
   invoke SendMessage, hDsm, LVM_SETTEXTBKCOLOR,0,00DEF5F3h
   invoke SendMessage, hDsm, LVM_SETEXTENDEDLISTVIEWSTYLE, 0,\
                             LVS_EX_FULLROWSELECT or \
                             LVS_EX_GRIDLINES + LVS_EX_FLATSB
```

First, it sends a message changing the text color, and then changes the back color for the *ListView*, and then change the back color for text. Make sure the back color for the text is the same as the back color for the *ListView*. The last message, change the extended *ListView* style, add Gridlines & Flat Scrollbars & allow us to select the whole row instead of just the item.

[1] will improve it more later!

4.3 Final Words

In this chapter we saw how *ListView* does work, and we saw a simple example displaying almost most of the features we need (for now).

4.4 In the next chapter

We will:

- Go on a journey inside the PE
- Add the (Events-Reporter) code

Chapter 5

Journey Inside The PE

5.1 What is the PE?

A lot of tutorials are talking about PE, and a lot of them are good, but we must go into this topic as it really concerns our project. We will be focused on the PE from our view.

First we know the PE stands for *Portable Executable*. When we say this file is a portable executable it means that any win32 platform will recognize and uses this format regardless of the CPU platform. In other words, every win32 executable file[1] uses this format.

5.2 Understanding the PE

Any PE file consists of:

- DOS MZ header
- DOS Stub
- PE File signature
- PE Header
- PE Optional header
- Sections (Section 1, Section 2, Section...., Section n)

That is the general layout of any PE. To get to know these stuff better, let us first construct a simple program that loads any executable file the user select by the menu, and map it to memory. Looks like we have already discussed how to do it, so after combining the code from the previous chapters, we made a small empty program - let us call it PEInfo - that does this job. You can find it attached in the book sources. The file simply loads any file, map it to memory and save the offset of the memory. So how can this help us understanding the PE file?

5.2.1 DOS MZ Header (IMAGE_DOS_HEADER)

It is 64 bytes long header which is mainly used to enable the program to from DOS and thus DOS can recognize it as a valid executable file and then run the DOS STUB. It is the first component in the PE file.

[1] Except the VXD and the old 16-bit Dll.

The structure is defined as:

```
IMAGE_DOS_HEADER STRUCT
    e_magic WORD ?           // Magic number
    e_cblp WORD ?            // Bytes on last page of file
    e_cp WORD ?              // Pages in file
    e_crlc WORD ?            // Relocations
    e_cparhdr WORD ?         // Size of header in paragraphs
    e_minalloc WORD ?        // Min. extra paragraphs needed
    e_maxalloc WORD ?        // Max. extra paragraphs needed
    e_ss WORD ?              // Initial SS value
    e_sp WORD ?              // Initial SP value
    e_csum WORD ?            // Checksum of the file
    e_ip WORD ?              // Initial IP value
    e_cs WORD ?              // Initial CS value
    e_lfarlc WORD ?          // Address of relocation table
    e_ovno WORD ?            //
    e_res WORD 4 dup (?);    // Reserved words
    e_oemid WORD ?           // OEM identifier
    e_oeminfo WORD ?         // OEM information
    e_res2 WORD 10 dup (?)   // Reserved words
    e_lfanew DWORD ?         // File address of new exe header
  IMAGE_DOS_HEADER ENDS
```

This is the structure of the DOS MZ Header. Actually the important things are for us[2]:

- The e_magic WORD, which holds the Magic Number. This number is used to identify the file type. All DOS compatible executable files have the value of this field as 0x54AD which is in ASCII represents the characters 'MZ' that's why we call it (DOS MZ Header).
- The e_lfanew field. This field which is a 4-byte offset to the PE Header. So, we use it to locate the PE Header.

5.2.2 DOS Stub

DOS Stub is actually a program that is used to run in MS-DOS. This program typically does nothing but output a simple line of text telling us that the program cannot be run in DOS. (though it is not our main interest, but actually DOS Stub is a dos program called WINSTUB.EXE, when building a program the compiler links this default stub to the main program. This behavior can be overridden by the linker)

5.2.3 PE File Signature

PE File Signature (for win32 files) is a DWORD == IMAGE_NT_SIGNATURE == 0x00004550 Which is represented in ASCII as 'PE',0x00,0x00

So, for a valid PE we must find this signature which as the DWORD at (OffsetOfFileInMem + e_lfanew).

[2]Notice: Offset of the mapped file + the value of e_lfanew == Start of the PE file header

5.2.4 PE Header (*IMAGE_NT_HEADERS*)

Now, the time for PE Header. Let us refresh our memory, PE Header is located at (e_lfanew Bytes) from the start of the PE. That's cool, but what can we know about this structure?

```
IMAGE_NT_HEADERS STRUCT
   Signature  dd  ?
   FileHeader  IMAGE_FILE_HEADER  <>
   OptionalHeader  IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Signature

We talked about about it (PE File Signature).

File Header (*IMAGE_FILE_HEADER*)

This structure - as we will see in details - contains general information about the layout (physical layout) of the PE.

```
IMAGE_FILE_HEADER STRUCT
   Machine               WORD   ?   ; Machine type (Alpha/Motorola/...)
                                      (0x014C == I386)
   NumberOfSections      WORD   ?   ; Number of sections in the file
   TimeDateStamp         dd     ?   ; The time that this file was created.
                                       This field holds the number of seconds
                                       since December 31st, 1969, at 4:00 P.M.
   PointerToSymbolTable  dd     ?   ; The file offset of the COFF symbol table.
                                       This field is only used in OBJ files and
                                       PE files with COFF debug information.
   NumberOfSymbols       dd     ?   ; The number of symbols in the
                                      COFF symbol table
   SizeOfOptionalHeader  WORD   ?   ; Size of the OptionalHeader structure
   Characteristics       WORD   ?   ; flags of the file (exe/dll/system file/..)
IMAGE_FILE_HEADER ENDS
```

The *NumberOfSections* is important to us, because we must know this value when we're walking through the sections table. The section table is like an array of structures, each one contains the information necessary for the section. So, if there is n sections, there will be n members of the array.

OptionalHeader (*IMAGE_OPTIONAL_HEADER32*)

Well, first do not get deceived by the name (Optional), this header is not optional in fact, it is there in every PE file. The optional header contains most of the information we need about the executable image, such as the initial stack size, the EP of the program, preferred base address, version of the operation system, information about the alignment of the section, and so on.

```
IMAGE_OPTIONAL_HEADER32 STRUCT
  Magic                         WORD    ? ; 2 bytes identifying the state
                                            of the file.
  MajorLinkerVersion            BYTE    ? ; Linker major version number.
  MinorLinkerVersion            BYTE    ? ; Linker minor version number.
  SizeOfCode                    DWORD   ? ; Size of the code section OR Sum of all
                                            code sections (multiple sections).
  SizeOfInitializedData         DWORD   ? ; Size of the initialized data OR ....
                                            multiple data sections.
  SizeOfUninitializedData       DWORD   ? ; Size of the uninitialized data section (BSS)
                                            OR ... multiple BBS sections.
  AddressOfEntryPoint           DWORD   ? ; Address of entry point (RVA of the
                                            1st instruction to be executed)
  BaseOfCode                    DWORD   ? ; Address (RVA) of beginning of code section.
  BaseOfData                    DWORD   ? ; Address (RVA) of beginning of data section.
  ImageBase                     DWORD   ? ; The *preferred* load address of the file
                                            (default is 0x00400000).
  SectionAlignment              DWORD   ? ; Alignment (in bytes) of sections when
                                            loaded into memory.
  FileAlignment                 DWORD   ? ; Alignment (in bytes) of sections in the
                                            file (multiplies of 512 bytes).
  MajorOperatingSystemVersion   WORD    ? ; Major version number of required OS.
  MinorOperatingSystemVersion   WORD    ? ; Minor version number of required OS.
  MajorImageVersion             WORD    ? ; Major version number of image.
  MinorImageVersion             WORD    ? ; Minor version number of image.
  MajorSubsystemVersion         WORD    ? ; Major version number of subsystem.
  MinorSubsystemVersion         WORD    ? ; Minor version number of subsystem.
  Win32VersionValue             DWORD   ? ; Dunno! But I guess for future use.
  SizeOfImage                   DWORD   ? ; Total size of the PE image in memory
                                            (All Headers & Sections aligned to
                                             SectionAlignment).
  SizeOfHeaders                 DWORD   ? ; Size of all headers & section table.
                                            (The file offset of the first section
                                             in the PE file)
  CheckSum                      DWORD   ? ; Image file checksum.
                                            (computing algorithm is in IMAGHELP.DLL)
  Subsystem                     WORD    ? ; Target subsystem of the PE file.
                                            (Mostly GUI & CUI)
  DllCharacteristics            WORD    ? ; Flags used to indicate if a DLL image
                                            includes EPs.
  SizeOfStackReserve            DWORD   ? ; Size of stack to reserve.
  SizeOfStackCommit             DWORD   ? ; Size of stack to commit.
  SizeOfHeapReserve             DWORD   ? ; Size of local heap space to reserve.
  SizeOfHeapCommit              DWORD   ? ; Size of local heap space to commit.
  LoaderFlags                   DWORD   ? ; Choose to (break/debug/run normally
                                            (default)) on load.
  NumberOfRvaAndSizes           DWORD   ? ; The length of the DataDirectory
                                            array that follows.
  DataDirectory                 IMAGE_DATA_DIRECTORY IMAGE_NUMBEROF_DIRECTORY_ENTRIES dup(<>)
IMAGE_OPTIONAL_HEADER32 ENDS
```

You might feel confused here, but do not worry, the important pars will be discussed with more details, just relax, read slowly, see the code example (when it is mentioned) and you will be fine.

First, we got to know about something very important.

RVA (Relative Virtual Address)

PE format uses this concept of the so-called RVA. So, what is RVA? RVA stands for Relative Virtual Address, it is used to describe the memory address if you do not know the base address. I assume that you know what offset means, RVA is the same as an offset BUT it is relative to a point in the virtual address memory, not a file.

Example:

Let's say, a program loads at (ImageBase) 0x00400000 at the Virtual Address (VA) memory, and start execution at the Virtual Address (VA) 0x00402000 then we can say that this program starts execution at the RVA 0x2000. As its name, RVA is Relative to the Virtual Address of the file. In other words, RVA is the value we need to add to the ImageBase to get the memory address.

Example:

The Program loads with ImageBase = 0x00400000 and RVA == 0x1234 then the actual execution of the program starts at (0x00400000 + 0x1234 == 0x00401234).

You may wonder why is this confusion? Well, because sections are not necessarily aligned the same as the loaded image of the file is. Meaning, sections of the file are aligned to 512-Byte multiplies (0x200) while the loaded image is aligned to 4096-Byte multiplies (0x1000). (Depends on SectionAlignment & FileAlignment). So, to get any information on specific RVA for a PE file, you must calculate the offset like this:

Suppose u have the RVA 0x1234 and you need to know the offset in the file, you check each section, so you find one section aligned to 0x1000 and 0x20000 bytes long. So, the offset (in this section) would be:

```
RVA - SectionAlignment  =  0x1234 - 0x1000 = 0x234
```

You will get used to this with more practice so do not worry if you are still confused.

We stopped at this member of the IMAGE_OPTIONAL_HEADER this member is really important. It contains an array (16 arrays) of the IMAGE_DATA_DIRECTORY structure.

```
IMAGE_DATA_DIRECTORY STRUCT
  VirtualAddress    DWORD      ? ;  RVA of the location of the
                                      directory.
  isize             DWORD      ? ;  Size of the directory.
IMAGE_DATA_DIRECTORY ENDS
```

Directories as defined in WINNT.H

```
// Directory Entries

// Export Directory
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0
// Import Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1
// Resource Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2
// Exception Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3
// Security Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6
// Description String
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7
// Machine Value (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8
// TLS Directory
#define IMAGE_DIRECTORY_ENTRY_TLS            9
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10
//
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11
//
#define IMAGE_DIRECTORY_ENTRY_IAT            12
```

Followed by a null IMAGE_DATA_DIRECTORY structure.

Sections (Header/Data)

One important thing to learn, is sections. Section table begins right after the optional header. The section table is a collection of section headers each is 40 bytes long. The purpose of the section header is to tell the loader where data, code, etc in the PE file should be mapped to memory. The number of sections in the section table is defined in the PE Header at the member *NumberOfSections*.

Each section consists of:

Header: Contains the section description and it is of the type IMAGE_SECTION_HEADER.

```
IMAGE_SIZEOF_SHORT_NAME equ 8

IMAGE_SECTION_HEADER STRUCT
   Name1                   db IMAGE_SIZEOF_SHORT_NAME dup(?)
   union Misc
      PhysicalAddress      dd ?
      VirtualSize          dd ?
   ends
   VirtualAddress          dd ?
   SizeOfRawData           dd ?
   PointerToRawData        dd ?
   PointerToRelocations    dd ?
   PointerToLinenumbers    dd ?
   NumberOfRelocations     dw ?
   NumberOfLinenumbers     dw ?
   Characteristics         dd ?
IMAGE_SECTION_HEADER ENDS
```

- **Name1 (name):**
  It's simply the name of the section (ANSI Name). Length is 8 bytes. This name is just a label, you can even leave it blank and it can be not null terminated!

- **PhysicalAddress:**
  Dunno exactly, some linkers just put 0 and the file works just fine!

- **VirtualSize:**
  Size of the file when it's mapped to memory. Must be multiple of 4096.

- **VirtualAddress:**
  The RVA of the file where it should be mapped to memory. (If ImageBase is 0x00400000 and this field is 0x1000 the file would be loaded at 0x0401000).

- **SizeOfRawData:**
  The size of the section, rounded to the next multiply of the FileAlignment, it's used by the system to know how many bytes should it map to the memory (multiple of 512).

- **PointerToRawData:**
  The offset in the file of the beginning of the section.

- **PointerToRelocations:**
  Not used.

- **PointerToLinenumbers:**
  Not used.

- **NumberOfRelocations:**
  Not used.
- **NumberOfLinenumbers:**
  File-based offset of the line number table. It's usually used for debugging purposes and usually set to 0.
- **Characteristics:** Flags for section characteristics as follow:

```
0x00000020 Code section
0x00000040 Initialized data section
0x00000080 Uninitialized data section
0x04000000 Section cannot be cached
0x08000000 Section is not pageable
0x10000000 Section is shared
0x20000000 Executable section
0x40000000 Readable section
0x80000000 Writable section
```

Data: After the section header comes the section data, in the file, they are aligned to the FileAlignment bytes, and in the order of their RVAs. While in memory, the sections are aligned to the SectionAlignment bytes.

General Information: There are many kinds of sections, depending on what's they contain.

Code Section: The code section must have the (0x00000020) flag. The AddressOfEntryPoint points to a location somewhere in this section. The BaseOfCode points to the start of this section (or somewhere in it). Mostly, this code contains nothing but executable code. Mostly, Code Section names are ('.text' - '.code' - 'AUTO').

Data Section: The initialized data section (the initialized static variables) e.g. (static int a = 10), it should have the (0x00000040) flag also (0x40000000 & 0x80000000) flags as well . The section is in the range (BaseOfData + SizeOfInitializedData). Mostly, Data Section names are ('.data' - '.idata' - 'DATA').

BSS Section: The uninitialized data section (the uninitialized static variables) e.g. (static int a), mostly like the the initialized data section except its PointerToRawData should be 0 this tells that its content is not stored in the file (not initialized). It also should have the (0x00000080) flag instead of (0x00000040). The length should be 'SizeOfUninitializedData'. Mostly BSS Section names are ('.bss' - 'BSS').

Imports Section: There is a LOT of things to say about this, so I put it in an individual part.
I think this information about the PE is enough for now, as u will understand more n more in the coming parts when we see how things work with examples. You should also read more tutorials if necessary, like Iczelion's [2] and Luevelsmeyer [4].

5.3 Checking for a valid PE

Okay, now the fun part begins. How can we check for a valid PE? As we saw above, a valid PE has two signatures that must be found on every PE, which are the DOS Header Signature ('MZ') and the PE Header Signature ('PE',0,0).

- We load the file n map it to memory.
- Starting from the first byte of the mapped file, we check for the DOS Header Signature.

```
  mov esi, FileOffset
  assume  esi: ptr IMAGE_DOS_HEADER
; esi is a IMAGE_DOS_HEADER structure
;=========================================
; Check for valid 'MZ' Signature
;=========================================
cmp word ptr [esi].e_magic, IMAGE_DOS_SIGNATURE
; Is 'MZ' Signature Is Valid?
jz @f
mov eax, -1 ; return -1 (Err)
ret
@@:
```

I think the above code is straight and simple:

- esi points to the 1st byte of the mapped file
- Assume that esi is an IMAGE_DOS_HEADER structure
- check the member e_magic for the value of IMAGE_DOS_SIGNATURE ('MZ') if they are identical then we passed the first check.

```
;=========================================
; Check for valid 'PE' Signature
;=========================================
add esi, [esi].e_lfanew
; esi = FileOffset + SizeOfDOSHeader == PE Header
mov NTHeaderOffset, esi
assume esi:ptr IMAGE_NT_HEADERS
cmp [esi].Signature, IMAGE_NT_SIGNATURE
; Is 'PE' Signature Is Valid?
jz @f
mov eax, -1 ; return -1 (Err)
ret
@@:
xor eax, eax
ret
```

As you now understand:

- esi (that held the offset of the 1st byte of the mapped file) get increased by (e_lfanew) bytes, which means (Start of the file + SizeOfDOSHeader == The start of the PE Header)
- Store the address of the PE Header for later access
- Assume that esi is an IMAGE_NT_HEADERS structure
- Check the member Signature for the value of IMAGE_NT_SIGNATURE ('PE,0,0') if they are identical then we passed the second check.

Now we know how to check for a valid PE file, see the source for more details.

5.4 Getting the PE Sections

As you know, sections are located right after the optional header (Right after the IMAGE_NT_HEADERS). It means, the first header should be located at (Offset of IMAGE_NT_HEADERS + SIZEOF IMAGE_NT_HEADERS).

So , simply you should get the *NumberOfSections* and then starting from the IMAGE_SECTION_HEADER you loop through the sections, and getting each one information. Something like this:

```
LOCAL pName:DWORD, dwVirtualAddress:DWORD, dwVirtualSize:DWORD, \
      dwPhysicalAddress:DWORD,dwPhysicalSize:DWORD, \
      dwCharacteristics:DWORD
```

These DWORDs are there to temporary the section information. Next we have some more explanations.

```
mov esi, NTHeaderOffset                ; esi points to the Offset
                                         of IMAGE_NT_HEADER structure
assume esi: ptr IMAGE_NT_HEADERS
add esi, sizeof IMAGE_NT_HEADERS       ; esi points to the first section
mov SectionsOffset, esi                ; Save the 1st section offset
assume esi: ptr IMAGE_SECTION_HEADER
xor ecx, ecx
@@:
```

The above code as it shows, gets (and stores) the offset of the 1st section. And this is done by getting the offset of the offset of IMAGE_NT_HEADERS and adding the SIZEOF IMAGE_NT_HEADERS to it. Now we have the offset of the 1st section. Let's continue.

```
xor ebx, ebx
mov bx, word ptr [esi].Misc.VirtualSize    ; VirtualSize ( + 0x08 )
mov dwVirtualSize, ebx                     ;
mov bx, word ptr [esi].VirtualAddress      ; VirtualAddress ( + 0x0C )
mov dwVirtualAddress, ebx                  ;
mov bx, word ptr [esi].SizeOfRawData       ; PhysicalSize ( + 0x10 )
mov dwPhysicalAddress, ebx                 ;
mov bx, word ptr [esi].PointerToRawData    ; PhysicalOffset ( + 0x14 )
mov dwPhysicalSize, ebx                    ;
mov ebx,  [esi].Characteristics            ; Characteristics
mov dwCharacteristics, ebx                 ;
lea ebx, [esi].Name1                       ; Name of the section
mov pName, ebx                             ;
add esi, sizeof IMAGE_SECTION_HEADER       ; Next Section
push ecx
```

Loop through the section header, save the important information for us (VirtualSize & VirtualAddress & PhysicalSize & PhysicalOffset & Section Name). After this, displaying the sections comes in place, before we talk about this Please See The Source Code Now so you can see the procedure AddEvent

```
AddEvent proc  pszEvent:DWORD
   ;=====================================================
   ; pszEvent: Pointer to the text we wanna insert
   ;=====================================================
   invoke SendMessage,hEventHandler,LB_ADDSTRING,0,pszEvent
   ret
AddEvent endp
```

Simply takes one parameter, pointer to a string to be inserted at the ListBox at the bottom, which we call (the ListBox) the EventHandler as it is the responsible for displaying the events as they occur. Anyway, back to the main topic, after getting the sections information it is time to display them on the screen, so we use the (AddEvent procedure to display the sections information) (See The Source Code For Details)

5.5 What is the EP?

EP or the Entry Point of the PE is simply the address where the execution of the code starts. It means, the first byte of code that will run is the first byte at the EP.

5.6 Getting the EP?

If you remember well *AddressOfEntryPoint* member is located at the *Optional Header*. Which is actually the RVA of the Entry Point. We can easily get this value, but wut if we want to get the offset in the file itself? Remember when we talked about RVA? it is time to practice a little on that. First, to get the RVA of the Entry Point:

```
GetEntryPointRVA proc
;=========================================================
; Gets the RVA of the EntryPoint (AddressOfEntryPoint)
;=========================================================
   mov esi, NTHeaderOffset
   assume esi: ptr IMAGE_NT_HEADERS
   mov eax, [esi].OptionalHeader.AddressOfEntryPoint
   mov EntryPointRVA, eax
   ret
GetEntryPointRVA endp
```

Pretty straight code - let me add some comments. It starts at the PE Header (IMAGE_NT_HEADERS), and access the member AddressOfEntryPoint at the Optional Header and store this value for later access.

5.7 Converting from RVA to Offset

EP or the Entry Point of the PE is simply the address where the execution of the code starts. It means, the first byte of code that will run is the first byte at the EP. Now, let us get back to the RVA part. How can we convert RVA to offset ? Let us have a look at the following code and then understand what it does:

```
RVAToOffset proc RVA:DWORD
   mov edx, SectionsOffset
   assume edx: ptr IMAGE_SECTION_HEADER
   mov ecx, NumOfSections
   mov edi, RVA
   .while ecx > 0 ; Loop through all the sections
      .if edi >=  [edx].VirtualAddress ;
         mov eax,[edx].VirtualAddress ;
         add eax,[edx].SizeOfRawData ;
            .if edi<eax ; Is the address in this section?
               mov eax,[edx].VirtualAddress ;
               sub edi,eax ; edi == Section RVA - Our RVA
               mov eax,[edx].PointerToRawData ;
               add eax,edi ; eax == file offset
               ret
            .endif
      .endif
      add edx,sizeof IMAGE_SECTION_HEADER
      dec ecx
   .endw
   mov eax,edi
   ret
RVAToOffset endp
```

The above function which takes a parameter (the RVA) and returns in eax the offset. So how it works? First, starting at the first section (remember? Offset of IMAGE_NT_HEADERS + SizeOf IMAGE_NT_HEADERS), and looping through each section (As in NumOfSections or (NumberOfSections)), so in each section:

- Compare the RVA we wanna convert with the VirtualAddress of the section, if it is less then the RVA is not in this section, so it moves to the next section. If the RVA is greater than or equal to the VirtualAddress of the section, then it can be in this section, so continue.
- Compare the RVA with the sum of (VirtualAddress + SizeOfRawData of the section), if the RVA is greater, then it cannot be in this section, so move to the next section. If the RVA is less than the sum of the VirtualAddress and SizeOfRawData, then we found the correct section.
- Simply we can say that we loop through each section and check for the section that fulfill these conditions: (VirtualAddress + SizeOfRawData) > RVA >= VirtualAddress
- We know now which section has this RVA, so to convert the RVA to offset we do this. (Subtract the RVA from the VirtualAddress of the section) (Add the PointerToRawData) This way we get the offset.
  Offset = (RVA-VirtualAddress) + PointerToRawData
- Finally it returns the offset in eax

5.8 What is the Import Table?

Well actually this is not a simple question with a simple answer, we are going to understand the Import Table in this topic, but take care as it needs attention as it is long and not for beginners!

5.8.1 The meaning of 'Import' function

First, let us stop at the word (Import).. What does it mean? Well, basically it means it is a function which is somewhere (in a DLL) outside the callers executable BUT called by it. When the compiler finds a call to such a function it will actually know nothing about this function so it will normally just output a normal (Call) instruction to its symbol (the address that the compiler will have to fix later, as it usually does with any external symbol).

After that the linker uses a kind of an import library, to look up from which DLL this symbol is imported, produces stubs for each imported function, each one consists of a (Jmp) instruction that will jump to an address . In some other basic words, you can say that it is an address of a function inside a DLL (export directory) which is looked up and fixed by the linker.

5.8.2 How can we access the Import Table?

Well, let us remember this:

```
IMAGE_NT_HEADERS STRUCT
   Signature         dd   ?
   FileHeader        IMAGE_FILE_HEADER  <>
   OptionalHeader    IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

And the optional header:

```
IMAGE_OPTIONAL_HEADER32 STRUCT
   .......
   .......
   .......
   DataDirectory   IMAGE_DATA_DIRECTORY
                   IMAGE_NUMBEROF_DIRECTORY_ENTRIES dup(<>)
IMAGE_OPTIONAL_HEADER32 ENDS
```

Can you see the point? As we said before, DataDirectory member is an array of the IMAGE_DATA_DIRECTORY structure.

```
0      Export symbols
1      Import symbols
...      ...
```

The Import symbols is the second entry of the DataDirectory, so how to get to it? Actually how to get to any member of the DataDirectory array? Simply you (starting from the PE Header) get the data directory address from the OptionalHeader, add n(SizeOf IMAGE_DATA_DIRECTORY) where (n) is the index of member you want (e.g. 1 for Import Symbols), and you will be at the start of the IMAGE_DATA_DIRECTORY structure for the data structure you want. We said before that IMAGE_DATA_DIRECTORY structure:

```
IMAGE_DATA_DIRECTORY STRUCT
   VirtualAddress    DWORD   ? ;  RVA of the location of the directory.
   isize             DWORD   ? ;  Size of the directory.
IMAGE_DATA_DIRECTORY ENDS
```

Is 8 Bytes long structure, the first 4 bytes (DWORD) is the RVA of the directory location, where the last 4 bytes (DWORD) are for the size of the directory. So, by getting the RVA of the location of the (Import Symbols) directory, we can land on the start of the Import Table.

5.8.3 So what is the Import Table?

Actually, the import table is an array of IMAGE_IMPORT_DESCRIPTOR structures, each of these structures has information about a DLL that the PE is importing symbols from. Which means, if you have a program that is importing functions from 5 different DLLs, there will be 5 arrays of this structure, and at the end this array is terminated by an IMAGE_IMPORT_DESCRIPTOR contains only zeros. Let us understand more.

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
   union
      Characteristics  dd ?
      OriginalFirstThunk  dd ?
   ends
   TimeDateStamp  dd ?
   ForwarderChain  dd ?
   Name1  dd ?
FirstThunk  dd ?
IMAGE_IMPORT_DESCRIPTOR ENDS
```

- **OriginalFirstThunk:** RVA for a (0-terminated) array of RVAs each points to a structure (IMAGE_THUNK_DATA), describing an imported function.
- **IMAGE_THUNK_DATA:** (RVA of IMAGE_IMPORT_BY_NAME). Hint: which is the index of the function in the export table of the DLL though this value is not very important and can be ignored (some linkers do).
- **Name:** The name of the of the import function.
- **Name1:** RVA to the name of the DLL (A pointer to the name of the DLL) which is ASCII string.
- **FirstThunk:** It may confuse you but it is very similar to the OriginalFirstThunk, contains RVA to an array of IMAGE_THUNK_DATA structures (just notice, it is a DIFFERENT array).

Let us stop the confusion about the OriginalFirstThunk and the FirstThunk. Imaging you have two arrays, filled with RVAs of IMAGE_THUNK_DATA (or from now on IMAGE_IMPORT_BY_NAME), structures, and the both arrays contain exactly the same RVAs (Like one is the copy of the other). Now, assign the first RVA of the first array to the OriginalFirstThunk and the first RVA of the second array to the FirstThunk. You should understand it now.

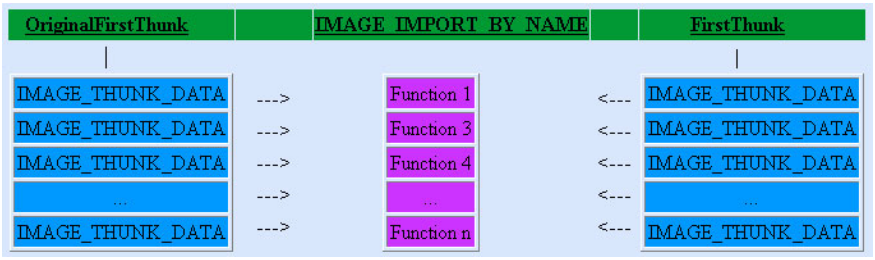


Figure 5.1: This table is to help you understand the OriginalFirstThunk & the FirstThunk

A silly[3] question: Why are there two identical arrays of IMAGE_THUNK_DATAs ? At run time, programs do not need the names of the imported functions, but actually they need the ADDRESSES of the functions. The loader will look up each of the imported symbol at the export directory of the DLL, and then replace the IMAGE_THUNK_DATA in the FirstThunk array, with the linear address of the DLL's Entry Point. While the OriginalFirstThunk remains untouched so we can look up the imported names via this list.

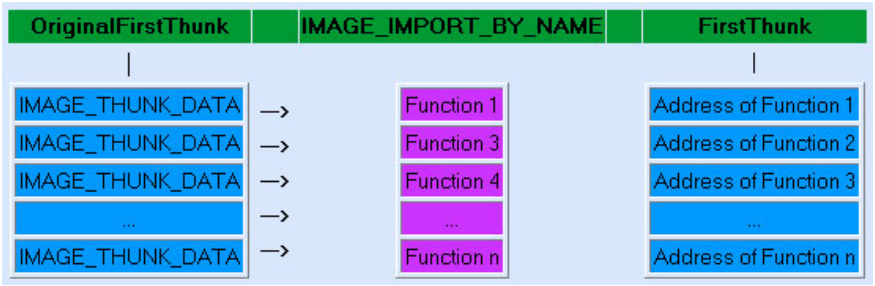


Figure 5.2: FirstThunk changed while OriginalFirstThunk is untouched

Well, if you are still confused, please read again, as we are not finished yet. One thing to mention is, it is not necessary that all functions are imported by their names! Some functions are imported by the ordinal only. That means you can not call this function by the name but you can call them by position. In such imports, there is no IMAGE_IMPORT_BY_NAME structure for this function. And the IMAGE_THUNK_DATA will contain the ordinal of the function in the low word and the most significant bit (MSB) is set to 1. (Micro$oft has come with the constant IMAGE_ORDINAL_FLAG32 = 0x80000000 which checks for the MSB in a DWORD. (e.g. test dwCheckMe, IMAGE_ORDINAL_FLAG32). Practice will help you get used to it, so reading the next part will be nice.

[3]It is not a silly question...

5.9 Getting the PE Imports

This part should clarify things that were confusing to you. What should we do to get a list of all the imports in a specific PE ? Basically what we should is:

- Go to the the first IMAGE_IMPORT_DESCRIPTOR structure
- Check the value of OriginalFirstThunk, if it is not zero, we get the RVA (in the OriginalFirstThunk) to the RVAs array (of IMAGE_IMPORT_BY_NAME). If it is zero, then we use the value of FirstThunk instead.
- Check every member in the array with the IMAGE_ORDINAL_FLAG32. If the MSB is 1 then this function is exported by ordinal and thus we should extract the ordinal of it from the low word of this member.
- If the MSB is 0 then this as an RVA to the IMAGE_IMPORT_BY_NAME, we go to it, skip two bytes (the Hint) and we'll be on the name of the function.
- Go to the next member at the array, get the name, and so on, till we get to the end of the array (null terminated), and we finished all the functions from one DLL.
- Go to the next IMAGE_IMPORT_DESCRIPTOR and do the same. Loop, till we get to the end of the array (zero-member).

Let us dive into the code:

```
mov esi, ImportSection ; We're at the first
                           ; IMAGE_IMPORT_DESCRIPTOR structure
assume esi: ptr IMAGE_IMPORT_DESCRIPTOR
.while [esi].FirstThunk != 0
    || [esi].OriginalFirstThunk != 0
    || [esi].TimeDateStamp != 0  ; empty structure
    || [esi].ForwarderChain != 0 ; OR NULL structure
   ....
   ....
   ....
   add esi, sizeof IMAGE_IMPORT_DESCRIPTOR
            ; Next IMAGE_IMPORT_DESCRIPTOR
.endw
```

First, we begin at the start of the Import Section and loop through each (IMAGE_IMPORT_DESCRIPTOR structure), checking if the structure is not (zero filled) the end of the array. If this is not a zero-filled array then we continue like this:

```
mov eax, [esi].Name1 ; Get the RVA of DLL name
```

We save this for later use (see the code to know more how we used TreeView control to insert data about imports).

```
.if [esi].OriginalFirstThunk == 0       ; Is OriginalFirstThunk = 0 ?
   mov eax, [esi].FirstThunk            ; Get FirstThunk
.else
   mov eax, [esi].OriginalFirstThunk    ; Get OriginalFirstThunk
.endif
```

Then we see if the OriginalFirstThunk is empty (zero) if so, we get the value of FirstThunk[4]. And continue:

```
.while dword ptr [edx] != 0
   test dword ptr [edx], IMAGE_ORDINAL_FLAG32   ; Imported By Ordinal?
   .if ZERO?
      invoke RVAToOffset, dword ptr [edx]       ; If not by Ordinal
      add eax, FileOffset
      mov edi, eax
      add edi, 2
      mov pName, edi
      ....
      ....
      add edx, 4
   .else    ; By Ordinal
      mov eax, dword ptr [edx]
      and eax, 0FFFFh                           ; Get the low WORD
      ....
      ....
      add edx, 4
   .endif
.endw
```

Check every RVA in the OriginalFirstThunk array if it is not zero check it against (0x08000000 or IMAGE_ORDINAL_FLAG32) to c if the function is imported by name or by ordinal. (check the MSB)

- If MSB = 0 (By Name) then the RVA is for an IMAGE_IMPORT_BY_NAME structure, so we add 2 bytes to skip the (Hint) member, and we're at the first byte of the name of the function.
- if MSB = 1 (By Ordinal) then, we take the value make a logical ADD to get only the low WORD of the DWORD, and we got the ordinal of the function.

I hope it was not hard to understand, but please have a look at the code for more details.

[4]But as a safe procedure we check the value of OriginalFirstThunk first

5.10 Final Words

Well, this chapter was LONG and not so easy for newbies, I hope we learned about about the PE. By this, I guess our disassembler is ready (for now) from the GUI part. Next chapters are the REAL work.

5.11 In the next chapter

We will:

- Introduction to Opcodes

Chapter 6

Introduction to Opcodes

6.1 What Are Opcodes?

First open the (Opcodes Sample - Part I) (OpcodesOne.exe) in your favorite disassembler[1] and let us take a look at what we see:

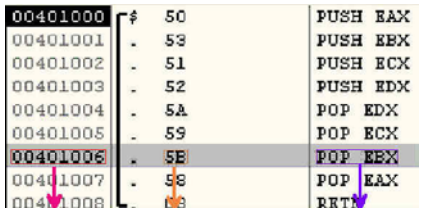


Figure 6.1: Opcodes in a disassembler

As you can see - at the leftmost column - there is the address of the byte disassembled[2]. In the middle column, there are the Opcodes, the rightmost column contains the instructions (Mnemonics). Each instruction (e.g. `Push eax`) is represented with a hex value (Opcode) in the file. When we are writing a program we write something like this:

```
push    eax
push    ebx
pop     eax
pop     ebx
```

What we wrote is the instruction itself (the mnemonic) but when we open the file in a hex viewer or in the disassembler, we do not find these instructions as we wrote them! Actually, the processor does NOT know what (`push eax`) means! But how can the processor understand our code?

The simple answer is: The assembler[3] converts the instructions from the Mnemonic to the Opcode[4].

```
(Mnemonics --> Assembler --> Opcodes )
(e.g. Push eax --> Assembler --> 0x50)
```

So does it mean that each mnemonic is another form (Alias) for an Opcode? Well. NO! For the specific case that (`push eax`) is ALIAS for `0x50` that is correct, but it is not the same for each and every opcode as we will see later. Actually, we will see later how a Mnemonic can have several Opcodes, an how an Opcode can have several Mnemonics! Just do not get confused, we will understand this later.

[1] We will use OllyDbg [7] for simplicity
[2] Starting from the EntryPoint
[3] e.g. ml.exe in MASM package [5]
[4] the hex values you see in the picture

6.2 Getting Familiar With Opcodes

Let us get more familiar with Opcodes and Mnemonics. Open the sample file (NOP.exe) in your Opcodes Samples directory with OllyDbg. You will soon see the following:

```
00401000 >/$  90            NOP <--- EntryPoint/Start of the code
00401001  |.  90            NOP
00401002  |.  90            NOP
00401003  |.  90            NOP
00401004  |.  90            NOP
00401005  |.  90            NOP
00401006  |.  90            NOP
00401007  |.  90            NOP
00401008  \.  C3            RETN
```

Now Double click on the first line[5], a dialog will appear to you:

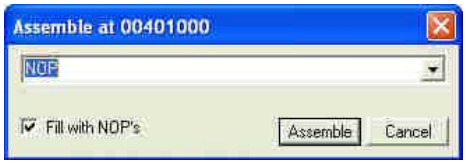


Figure 6.2: OpCode editor in OllyDbg

One of the nice features in OllyDbg is that assemble any mnemonic to opcode via this dialog. So let us write:

```
push eax
```

and press the '*Assemble*' button or just press Enter and close the dialog. You will see that OllyDbg recognized the mnemonic and converted it to the opcode.

```
00401000 >/$  50            PUSH EAX
```

[5]or click assemble, or press space bar

As you probably noticed `0x50` is the opcode for `push eax`. Now let us practice the opposite thing. Converting from the Opcode to the Mnemonic. In OllyDbg select the second line and press (`CTRL + E/Binary Edit`) the Binary Edit dialog will appear to you:

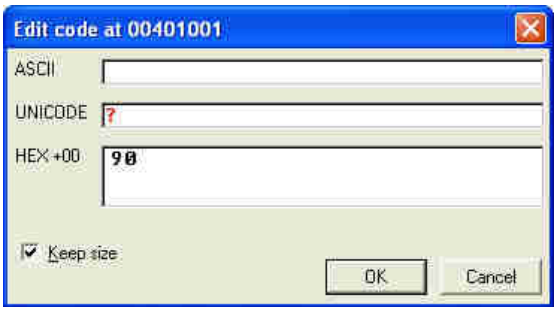


Figure 6.3: Binary Edit Dialog in OllyDbg.

Now change the value in hex (`0x90`) to the opcode that we know (`0x50`) and then press `OK`. Do you see what happend? OllyDbg recognized your Opcode and converted it to Mnemonic.

```
00401001  |.  50            PUSH EAX
```

If you want practice on some other opcodes till you get familiar with opcodes and mnemonics.

6.3 One Opcode Means One Mnemonic?

So is there only one mnemonic for an opcode? Well, remember the (`NOP.exe`) which OllyDbg disassembled as:

```
00401000 >/$  90            NOP
00401001  |.  90            NOP
00401002  |.  90            NOP
00401003  |.  90            NOP
00401004  |.  90            NOP
00401005  |.  90            NOP
00401006  |.  90            NOP
00401007  |.  90            NOP
00401008  \.  C3            RETN
```

Now open the source of the program and check it for your self!

```
.386
.model flat, stdcall  ;32 bit memory model
option casemap :none  ;case sensitive

.code

start:
   xchg eax, eax
   nop
   xchg eax, eax
   nop
   xchg eax, eax
   nop
   xchg eax, eax
   nop
   ret
end start
```

Surprised? Conclusion: Different Mnemonics can have the same Opcode! But not enough. Let us open the program (`Add eax, 1.exe`) in OllyDbg:

```
00401000 >/$  83C0 01         ADD EAX,1
00401003  |.  05 01000000     ADD EAX,1
00401008  \.  C3              RETN
```

Another surprise? Conclusion: Different Opcodes can have the same Mnemonic[6].

Another example:

```
00401000 >/$  03C0               ADD EAX,EAX
00401002  |.  01C0               ADD EAX,EAX
00401004  |.  02C0               ADD AL,AL
00401006  |.  00C0               ADD AL,AL
```

Anyway, we understood the basics about what Opcodes/Mnemonics are. Let us dig further.

[6]We will talk about structure groups later.

6.4 More About Opcodes

Now, let us know more about Opcodes. Let us open OllyDbg and load any file (e.g. `NOP.exe`), press `space` to open the assemble dialog. Let us enter:

- Add eax, 11223344
- Add eax, 12345678

We will see that OllyDbg disassembled these two instructions like this:

```
00401000 > $  05 44332211      ADD EAX,11223344
00401005   .  05 78563412      ADD EAX,12345678
```

It seems that (`05`) is responsible for the instruction (`Add eax, imm`) and the following (immediate) bytes, are the value to be added to `eax`. This immediate value, is[7] reversed. In other words, the bytes are ordered from right to left, not from left to right. So if we want to put the value (a hundred million times) in `eax`, we write (`mov eax, 10000000`) but the opcode generated is (`05 00000010`).

Now go and practice for a while on the (immediate) data. Remember when we said:

> Different Opcodes can have the same Mnemonic.

So what makes a specific opcode be chosen for a specific Mnemonic? Meaning: if we write a Mnemonic that can have more than one opcode, which opcode would be chosen and why?

Well the simple answer is: It is the assembler (like ml.exe in MASM) that does this job! It decides the optimal opcode and uses it, maybe it is not what you want but you know you can always choose wut opcode you want to use, in MASM you do this (`db 0x??, 0x??, 0x??, 0x??, .....`).

[7] compared to the way we write it in our life

6.5 Introduction To Intel Instruction Format

We must learn about the Intel Instruction Format in order to go on with our disassembler, so what is it? The General format for the instruction can be seen in figure 6.4.

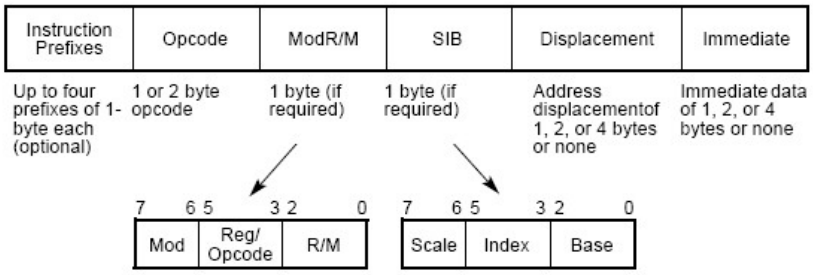


Figure 6.4: Intel Architecture Instruction Format

Intel instructions can vary in size[8] but they still have the same six groups. We need to understand each group first, and know it is purpose in order to be able to learn the sizes of the different instructions. And all these parts are optional except the opcode part.

6.5.1 Prefix (Optional)

The first part of Intel instruction, they change the behavior of the instruction in many ways:

- Change the default segment of the instruction:

```
2EHCS segment override prefix.
36HSS segment override prefix.
3EHDS segment override prefix.
26HES segment override prefix.
64HFS segment override prefix.
65HGS segment override prefix.
```

- Override the default size (of the machine-word):

```
Operand-size override, 66H
```

- Control loops in string operations:

```
F0HLOCK
F2HREPNE/REPNZ  (used only with string instructions).
F3HREP  (used only with string instructions).
F3HREPE/REPZ  (used only with string instructions).
```

[8] from 1 byte up to 14 bytes

- Override the Address size:

```
Address-size override, 67H
```

Operand-Size Prefix: When executing any instruction, the processor can address memory using either 16-bit or 32-bit address. Each instruction that access memory addresses has associated with it, the address size attribute of either the 16 or 32 bits. This attribute is determined by the instruction prefixes (and for other protected mode instructions bits in segment descriptor) .

Address-Size Prefix: Instructions that use the stack (e.g. `push eax`/`pop eax`) have address-size prefix which decides the use of 16 bits or 32 bits .

Let us take the (Operand Size Override) for example and see what it does. Open (`Empty.exe`) in OllyDbg, it is an empty program full of NOPs, open `binary change` dialog and enter:

```
50:
```

Press enter, you will see in OllyDbg

```
00401000   50            PUSH EAX
```

Now in the new line enter:

```
6650:
```

Press enter:

```
00401001    66:50        PUSH AX
```

Obviously you can see the difference! The operand (`eax` in this case) size changed from (32-Bit) to (16-Bit). Let us do one more example about prefixes so you get more familiar with them. Let us take the (Repeat Prefix) as another example and see what it does. Reload `Empty.exe` in OllyDbg (`CTRL+F2`), open `binary change` dialog and enter:

```
AC:
```

Press enter, you'll c in OllyDbg

```
00401003  |.  AC            LODS BYTE PTR DS:[ESI]
```

Now in the new line enter:

```
F2AC:
```

Press enter:

```
00401004  |.  F2:AC       REPNE LODS BYTE PTR DS:[ESI]
```

You can see the difference! Okay, now we know that if there is a Prefix then the instruction still needs the Opcode part. So it becomes like this:

```
[Prefix] [Code]
```

6.5.2 Code (Not Optional)

The operation code, which is the main instruction part, it comes after the optional prefixes. Actually, the code part is responsible for telling the processor which instruction to execute. This filed (code) contain bit fields that describes the size and the type of operand to expect. For example: The instruction (`NOT`). This instruction has the code byte `1111011x` where x is the bit responsible for specifying whether the operand is a BYTE or a DWORD.

- If x == 1 (the opcode is 11110111 == F7h)
  Then: F7:D0 == NOT EAX (DWORD)
- If x == 0 (the opcode is 11110110 == F6h)
  Then: F6:D0 == NOT AL (BYTE)

So, how can we get (`NOT AX`) (WORD)? Try to guess your self. Yes, you are right we must use Operand Size Prefix (`66h`):

```
66:F7D0 == NOT AX
```

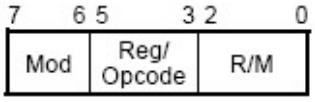
Let us take another example. The instruction (`OR`) that we use a lot. This instruction has the opcode (`000010xy`) where:

- The Bit (x) is responsible for specifying which which operands the source and the destination are.
- The Bit (y) is responsible for specifying the size of the operand.

We will talk more about opcode encoding later. The Code field can have different sizes. It can be one byte, two bytes or even 3 bytes[9].

6.5.3 ModR/M (00:000:000b)

If the instruction needs it, this part (`ModR/M`) that comes after the opcode, tells the processor which registers or memory locations to be used by the instruction. This member consists of three parts. Actually most the instructions that refer to an operand in memory have this prefix. The encoding of this prefix tells whether there is a (`SIB`) prefix or not.



Mod field: Occupies the the two most significant bits (`MSB`). Combines with the `r/m` field to form 32 possible values: eight registers and 24 addressing modes[10].

[9] Some SSE (And SSE2) [Streaming SIMD Extensions] instructions, which is mainly used in 3D-Graphics Modeling, can have code part more that two bytes! But we should not confuse our selves with this for now

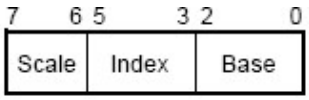
[10] Open ModRM I.exe and ModRM II.exe for some more details

opcode/reg field: Occupies the next three bits after the Mod field. Specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.

r/m field: Occupies the three least significant bits (`LSB`). Can specify a register as an operand or can be combined with the mod field to encode an addressing mode[11].

6.5.4 SIB

`SIB` stands for (Scale - Index register - Base register):



Scale: Occupies the two most significant bits (6-7) and specifies the scale factor.

Index: Occupies the next three bits after the Scale field, specifies the register number of the index register.

Base: Occupies the three least significant bits of the SIB byte, specifies the register number of the base register.

The idea of the `SIB` byte is to generate the [`Base + Scale * Index`] which can be added also to any displacement specified in the `ModR/M` Byte. In other words, `SIB` byte enable the use of complicated addresses like:

```
mov eax, dword ptr esi +ebx*4+00401000
```

The `SIB` byte is usually not present, only when the instruction needs the addressing format (`Base+Scale*Index`).

6.5.5 Displacement

When the Mod field at the `ModR/M` Byte is either (01 or 10) the displacement is part of the operand's address. Displacement comes right after the `ModR/M` & `SIB` byte. The size of the displacement depends on the `ModR/M` mod field.

6.5.6 Immediate

Immediate values are there when an instruction needs it as an operand, such as (`mov eax, 1000`). The immediate value is the last part of the instruction, like displacement, immediate value can be either a byte or a word.

[11] Open ModRM I.exe and ModRM II.exe for some more details

6.6 Final Words

In this chapter we learned the basic stuff about Opcodes and IA (Intel Architecture) Format, it may have been confusing to some of you, but we will learn more during the next chapters. In the next chapters we will understand each part of the instruction format with more details and examples.

6.7 In the next chapter

We will read:

- Everything about Prefixes

Chapter 7

Everything about Prefixes

7.1 More About Prefixes

In the previous chapter, we got introduced to the Intel Architecture Instruction Format, now it's time to really understand each part of the instruction and in this chapter we'll deal with Prefixes .Prefixes can be put in five groups depending on how do they affect our apps:

- Prefixes that specify the segment (2E - 36 - 3E - 26 - 64 - 65)
- Prefix that changes the default operand size (66)
- Prefix that changes the default address size (67)
- Prefixes that change the string operation n loops (F3 - F2)
- Prefix that controls the processor BUS (F0)

Some Facts About Prefixes: Let's know some facts about prefixes as this will help us getting a better view for the IA (Intel Architecture) Instruction itself.

- One opcode can have several prefix (Up to Four Prefixes)
- Each prefix is 1 byte size (It means prefixes maybe up to four bytes)
- Sometimes (as we'll c) may not be used by the instruction, in such cases the prefix is just ignored.

Now, let's understand each prefix type clearly.

7.2 Segments Override Prefixes

First let us quickly know what are segments. We know that everything we see (and don't see) is actually a sequence of bytes. The processor needs to organize its access to memory so it can handle this huge sequence of bytes. So, whenever a byte or more are being accessed, the processor uses a (byte address) to locate the byte or bytes in memory. The space or the range of bytes that can be addressed, is called the Address Space. Segments can be considered as a form of addressing, where the program can have many independent address spaces, these are called segments. The notation used to specify a byte address with the use of segments is:

```
Segment Registers: CS               : Code Segment
                   DS - ES - FS - GS: Data Segments
                   SS               : Stack Segment)
Segment (Register) : Byte-Address
```

Example for this: CS:[123456] –> This segment address identifies the byte located at the address 123456 in the segment that is pointed by the CS register. Let us not get out of the subject, we now got a quick idea about segments, so let us see what segments override prefixes do (it should be easy 2 guess).

```
-------------------------------------------------------------
00401000 >/$  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401002  |.  2E:8B00              MOV EAX,DWORD PTR CS:[EAX]
-------------------------------------------------------------
00401005  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401007  |.  36:8B00             MOV EAX,DWORD PTR SS:[EAX]
-------------------------------------------------------------
0040100A  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
0040100C  |.  3E:8B00             MOV EAX,DWORD PTR DS:[EAX]
-------------------------------------------------------------
0040100F  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401011  |.  26:8B00             MOV EAX,DWORD PTR ES:[EAX]
-------------------------------------------------------------
00401014  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401016  |.  64:8B00             MOV EAX,DWORD PTR FS:[EAX]
-------------------------------------------------------------
00401019  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
0040101B  |.  65:8B00             MOV EAX,DWORD PTR GS:[EAX]
-------------------------------------------------------------
```

You can see segment prefixes are responsible for choosing which segment register we want to access, the prefixes is then followed by a ModR/M byte (More about this in later chapter!) and a displacement (Later about this on a later chapter!). Also notice this (third example from above):

```
-------------------------------------------------------------
00401005  |.  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401007  |.  3E:8B00             MOV EAX,DWORD PTR DS:[EAX]
-------------------------------------------------------------
```

We can easily find that the prefix (3E) didn't override the default prefix, and this shows as we said at the first of this chapter that, if a prefix is not used (it's useless here) then it is just ignored by the processor. Please go and experience the use of segment prefixes in OllyDbg, use various examples, like:

```
00401000 > $  AC                  LODS BYTE PTR DS:[ESI]
00401001   .  2E:AC               LODS BYTE PTR CS:[ESI]
```

Try different opcodes, different sizes, and see how the prefixes can affect ur opcodes. Also don't forget to look at the Prefixes.exe file, included.

7.3 Operand-Size Prefix

Now, let's talk about a more used prefix, the Operand-Size Prefix. Let's first talk about the default operand size. There's is the 32-Bit operand size which is the default and there's the 16-bit operand size, these are the two default and only operand sizes. The processor is designed to support both operand sizes 32-bit and 16-bit, actually the only difference between the instruction that supports both operand sizes, is the Operand-Size Prefix at the start of the instruction. Operand size specifies the sizes of operands that the instruction will operate on.

So, when operand size is 16-bits, operands can be either 8-bits or 16-bits, when operand size is 32-bits operands can be either 8-bits or 32-bits. In other words, Operand Size Prefix 66h is used to choose the non-default operand size. Let's see some examples:

```
--------------------------------------
8BC0                    MOV EAX,EAX
--------------------------------------
```

This simple instruction (mov eax, eax) uses operand size of the 32-Bit size (as you can c it's accessing a 32-bits register to a 16-bits one!). So, in this instruction it means that the default operand size is the 32-bits size. Can you guess what would be the effect of the Operand-Size Prefix? (Which convert from 32-Bits to 16-Bits and vice versa :P) .Let's c if we add the 66h byte before this instruction. But did u ask ur self wut decides it the instruction (in operand-size 32-bits) will be either 8–bits or 32-bits? Well, to answer this we'll have to talk about opcode decoding which is not the time for it yet, so don't worry about this now.

```
----------------------------------
66:8BC0                 MOV AX, AX
----------------------------------
```

Can you see what happened? The instruction is the same but in a different operand size! Instead of the 32-Bits register (eax) there's the 16-bits register (ax). Please Have a look at (OperandSize.asm & OperandSize.exe) and of course Prefixes.exe. You can skip to the next part (Address-Size Operand) Directly if you don't wanna confuse yourself.

7.3.1 What's responsible for choosing the default Operand Size?

If you wondered what specifies whether the default Operand Size is 32-Bit or 16-Bit, Here's the answer, but u don't need to bother ur self with it so much. When we're coding a Win32 program, u can know that the default operand size is 32-Bit, U know why? That's because in the protected mode (Where Win32 Programs run) the Segment Descriptor defines the default operand (and address) size, and it's 32-Bit for Win32 Programs. How this default operand size is controlled via the segment descriptor? It's specified in the D (Default Size) Flag. When this flag is set (1) then 32-Bit operand (and address) size is selected, when the flag is cleared (0) then 16-Bit operand (and address) size is selected.

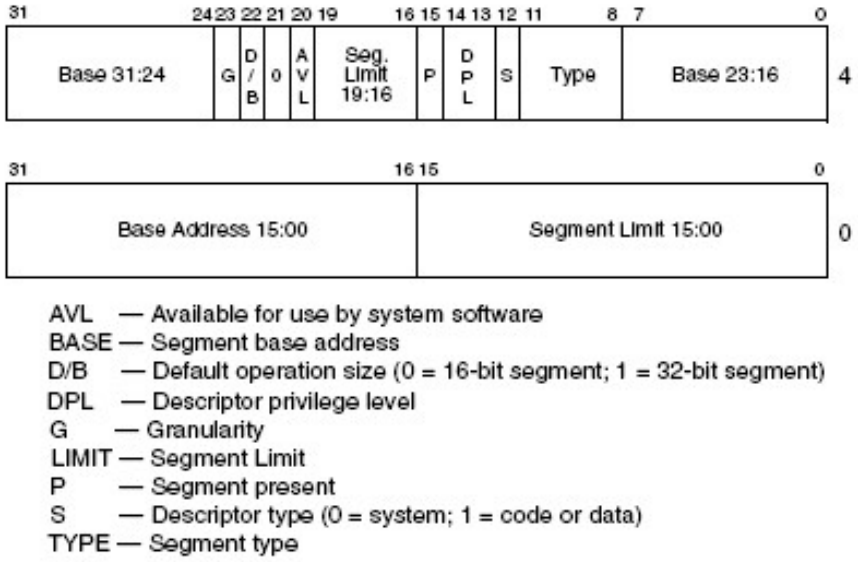


Figure 7.1: Segment Descriptor - (D) specifies the use of either 32-bit or 16-bit Operand Size.

7.4 Address-Size Prefix

Another important prefix here, the Address-Size Prefix (67h). Let's see this by example, let's see what the Address-Prefix do.

```
-----------------------------------------------------------------
00401000 >/$  8B00                MOV EAX,DWORD PTR DS:[EAX]
00401002  |.  67:8B00             MOV EAX,DWORD PTR DS:[BX+SI]
00401005  |.  8B01                MOV EAX,DWORD PTR DS:[ECX]
00401007  |.  67:8B01             MOV EAX,DWORD PTR DS:[BX+DI]
0040100A  |.  8B02                MOV EAX,DWORD PTR DS:[EDX]
0040100C  |.  67:8B02             MOV EAX,DWORD PTR SS:[BP+SI]
0040100F  |.  8B03                MOV EAX,DWORD PTR DS:[EBX]
00401011  |.  67:8B03             MOV EAX,DWORD PTR SS:[BP+DI]
-----------------------------------------------------------------
```

As you can see, the Address-Size is no more 32-bit (eax/ecx/edx/ebx) instead, there're (BS+SI/BX+DI/..) and so on. You must the ModR/M & SIB parts before you go on in this part, so I'll stop talking about the 67h (Address-Size Prefix) for now, and we will continue it for sure in later chapter, when we understand the necessary 2 instruction parts.

7.5 REP/REPNE Prefixes

If you have some experience with assembly especially with string operations/ instructions (e.g. movs/lods/scas/..), you'll for sure know what are (REPeat Prefixes). The REPeat Prefixes -if we can call them- can have three different forms and yet still two prefixes only! Confused? let's clarify it. Take a look at this table:

Repeat Prefix	Repeat Prefix	Term. Cond. 1	Term. Cond. 2
REP	F3	ECX=0	None
REPE/REPZ	F3	ECX=0	ZF=0
REPNE/REPNZ	F3	ECX=0	ZF=1

Table 7.1: REP/REPNE Prefixes

This table shows the difference three forms of the REPeat Prefixes, and shows that they only can have only two hex values. REPeat prefixes actually do nothing but repeat a string instruction till one of the condition (Look at the table above) is met, depending on which REPeat prefix we are using.

- **REP:** This Prefix works with these next string instructions: (INS - MOVS - OUTS - LODS - STOS)
- **REPE:** This Prefix works with these next string instructions: (CMPS - SCAS)
- **REPNE:** This Prefix works with these next string instructions: (CMPS - SCAS)

Now let's take a look at a quick example:

```
00401000 >/$  AD            LODS DWORD PTR DS:[ESI]
00401001  |.  F3:AD         REP LODS DWORD PTR DS:[ESI]
00401003  |.  F2:AD         REPNE LODS DWORD PTR DS:[ESI]
```

I think the example is obvious so no need to comment.

One more thing, let's know what's the difference between REPE & REPNE. To know this, we should look at these two prefixes as 8-Bits not as 1-Byte as well as understanding what (CMPS & SCAS) instructions do.

```
REPE   (F3) --->  1111 0011        Last Bit is (1)
REPNE  (F2) --->  1111 0010        Last Bit is (0)
```

Now leave this for now and let's see how (CMPS & SCAS) instructions work. The first one, compares a byte, word, dword in the source, with a byte, word, dword in the destination, and then sets the status flags in the EFLAGS according to the result. Similarly work the second one, but the destination is always the value in AL, AX, or EAX. So both instructions sets the status flag in the EFLAGS, and then the last bit of the REPeat Prefix is compared against this flag (ZF) if they are not the same no more repetition, terminate the instruction.

7.6 Bus LOCK Prefix

The last Prefix Bus LOCK Prefix (F0). Well, this prefix is not used a lot, as it's only used multiprocessor environment, it causes the processor's signal (LOCK#) to be asserted while executing an instruction, this signal (LOCK#) makes sure that the processor will have the exclusive use of the shared memory when the signal was asserted. The LOCK prefix works only on some instructions, especially those that use memory operands. (ADD - ADC - AND - BTC - BTR - BTS - CMPXCHG - DEC - INC - NEG - NOT - OR - SBB - SUB - XOR - XADD - XCHG) Well, to be honest, i don't have much info about the LOCK Prefix, maybe cause as i said it's only used in multiprocessor environment, but anyway i think this is enough for this instruction as we won't use it.

7.7 Final Words

In this chapter we -as I hope - could understand more about prefixes, it's true that we don't know 100% about them yet, but there're some few important tings that we'll know more in the next chapters, because i don't wanna mess the sequence of the tutorial.

7.8 In the next chapter

We will read:

- Everything about the [CODE] part in the Instruction format.

Chapter 8

Everything About [CODE] part I

On this part, we'll learn more about the [CODE] part in the IA32 Instruction Format, which -as we know- the second part of the instruction format. The [CODE] is the main part of the instruction, it's the only (Un-Optional) part of the instruction. To know more about that, we will study some 1-Byte Instructions (1-Byte Instruction means, this instruction has no additional bytes more than the [CODE] byte, e.g. No Prefix, No ModR/M, No SIB, and so on), but before that, let's know some -basic- info about [CODE] block.

8.1 Basics Of [CODE] Block

I assume by reading the tutorial till now that you're familiar with Hexadecimal & Binary Numbering systems, that's why I'm not gonna deeply talk about these two numbering systems. The reason why I'm talking about binary system is that I will try to explain a little bit, how the processor actually decode the instructions. The processor as we probably said doesn't know anything about what let's say (Push eax) mean! Actually the processor doesn't understand any mnemonic, it simply understand the instruction by decoding it, and this works like this:

- Most modern computers nowadays use the binary system
- The two values of binary system (0 & 1) are represented using two different voltage levels (Usually they are: 0v & +5v)
- The processor (I'm talking about Intel here) has a built-in decoding table which consists of a) The binary form of the instruction b) Rules of this instruction (e.g. addressing mode/addressing size/...)

We said that the processor has a decoding table, that means it requires a -signature- to be decoded. We can think about the [CODE] block as this signature, that tells the processor exactly what instruction to execute, and what kinds of rules should the instruction have. Let's have an example now, for one of the 1-Byte Opcode [CODE] block. One of the commonly used instructions is the (PUSH <reg>) instruction. Where <reg> can mean any of the 24 registers.

```
PUSH EAX   --> Let's study this instruction
```

The opcode for this mnemonic is 0x50, looking at the opcode as a single byte doesn't help us a lot, so let's look at the binary form of this opcode.

```
0x50 == 01010000b
```

and then let's make the 8-Bits look like this

```
--> 01010:000 == [CODE]<rrr>
```

What we've done is that we grouped the leftmost 5-Bits together (CODE) and the the rightmost three bits together (register).

- The 5-Bits (01010) are the [CODE] block responsible for the instruction (PUSH <reg>)
- The 3-Bits (000) are the <rrr> or the register code for the register (eax)

Let's have a look at the registers table that is used:

```
Register Table:
rrr   8bit  16bit   32bit
000 :  AL  :  AX  :  EAX
001 :  CL  :  CX  :  ECX
010 :  DL  :  DX  :  EDX
011 :  BL  :  BX  :  EBX
100 :  AH  :  SP  :  ESP
101 :  CH  :  BP  :  EBP
110 :  DH  :  SI  :  ESI
111 :  BH  :  DI  :  EDI
```

Let's stick for now on the 32-Bit registers. We said that 01010:000 == PUSH EAX because:

- 01010 is the [CODE] block for the instruction (PUSH <reg>)
- 000 is the 3-bit equal (eax) in the register table above

Let's take a look at the following examples (we're talking about 32-Bit Mode):

```
01010[111] == PUSH EDI == 0x57
01010[101] == PUSH EBP == 0x55
01010[011] == PUSH EBX == 0x53
01010[110] == PUSH ESI == 0x56
```

So far we've covered the 32-Bit size 8 registers, still 16 registers. So why I kept on saying (we're talking about the 32-Bit mode)? Remember the prefixes chapter? When we talked about Operand-Size override prefix (0x66) the prefix is responsible from switching from 32-Bit size to 16-Bit size mode. So the above examples would be like this:

```
66:01010[111] == PUSH DI == 0x66:0x57
66:01010[101] == PUSH BP == 0x66:0x55
66:01010[011] == PUSH BX == 0x66:0x53
66:01010[110] == PUSH SI == 0x66:0x56
```

Now we've covered the 32-Bit size 8 registers + the 16-Bit size 8 registers, still 8 registers (later). Let's take a look at another 1-Byte instruction to play with.

```
POP <Reg> 01011rrr
Code Block: 01011  == 0x5?
Reg: rrr
01011[111] == POP EDI == 0x5F
01011[101] == POP EBP == 0x5D
01011[011] == POP EBX == 0x5B
01011[110] == POP ESI == 0x5E
```

And the 16-Bit registers:

```
66:01011[111] == POP DI == 0x66:0x5F
66:01011[101] == POP BP == 0x66:0x5D
66:01011[011] == POP BX == 0x66:0x5B
66:01011[110] == POP SI == 0x66:0x5E
```

I hope you've understood something about the [CODE] block, still some other stuff to learn about, but you must understand the above test before we get into the next parts.

You can go practice on your own with other 1-Byte instructions, or use the tool attached with this chapter. Still some -advanced- stuff about [CODE] we gotta know, and we -hopefully- will know, in the next chapter. Just get don't move on till you're finished with this chapter. I divided the [CODE] part into two chapters for some reasons that I find important (like having a short chapter, not to loose focus as we'll talk -shortly- about ModR/M & SIB n for some other reasons). So before you go on to the next chapter be sure you understand this one well.

8.2 Final Words

I think the real fun starts from this chapter and the next chapters, this chapter was like half-way in understanding the second part of the Intel Architecture Instruction Format [CODE], I tried to make it as easy to understand as possible.

8.3 In the next chapter

We will read:

- Finishing the [CODE] part in the Instruction format.

Chapter 9

Everything About [CODE] part II

In the previous chapter we learned the basics about the [CODE] part, and we saw how 1-Byte instruction can be decoded into 5-Bits [CODE] block and 3-Bits register (<reg>) code. In this chapter we'll talk more about the decoding of the [CODE] byte. Let's take a look at this opcode (In Binary as usual):

```
00401000 >/$  F7:D0              not     eax   (F7h == 11110111)
00401000 >/$  F6:D0              not     al    (F6h == 11110110)
```

Can you see? There is no Operand-Size Prefix (Actually we're dealing here with 8-Bit register 'al' not 16-bit), there's no change in the two above instructions but the [CODE] bye itself! Let's make things clear. In the previous chapter (Seven I) we saw how instructions like (push <reg>/pop <reg>) can be decoded like this: 00000 : 000 Leftmost 5-Bits are the instruction code, rightmost 3-bits are the register code. This is not always the case. (e.g. only one bye opcode with reg field can be decoded like this 5 : 3). The example above (not eax/not ax) shows that the rightmost bit chooses the operand size to work on.

```
If Leftmost bit == 1 then we're in 32-Bits mode
If leftmost bit == 0 then we're in 8-bits mode
```

The above example can be decoded like this: `1111011w`

```
Bit (w): Operand-Size  (1= 32-Bit while 0=8-Bit)
```

One more thing to know in [CODE] block decoding, is another bit, the one that's right before the (w) bit. This bit (called d bit) chooses the direction. In this chapter I'll give you an example only about the (d) bit, before we show a lot of examples in the next chapter (ModRM).

```
00401000 >    A1 00000000   mov     eax, dword ptr ds:[0]
00401000 >    A3 00000000   mov     dword ptr ds:[0], eax
```

the (00000000) dword doesn't concert us now, we'll talk about later in (chapter ten: immediates), let's focus on the first byte, the [CODE].

```
A1: In Binary format is:    10100001       (d bit == 0 ==  reg --> imm)
A3: In Binary format is:    10100011       (d bit == 1 ==  imm --> reg )
```

The above example show the decoding like this: `101000d1`

I hope this shows what does the bit d do. Now let's combine the both bits (d & w).

```
00401000 >  A0 00000000  mov al, byte ptr ds:[0]    ( d = 0 / w = 0 )
00401000 >  A1 00000000  mov eax, dword ptr ds:[0]  ( d = 0 / w = 1 )
00401000 >  A2 00000000  mov byte ptr ds:[0], al    ( d = 1 / w = 0 )
00401000 >  A3 00000000  mov dword ptr ds:[0], eax  ( d = 1 / w = 1 )
```

One more thing. How many times did you look at the disassembly of a program (let's say in OllyDbg) and saw something like:

```
(... byte     ptr [..] )
(... word     ptr [..] )
(... dword    ptr [..] )
```

Bit	0	1
d	reg –> imm	imm –> reg
w	Operand Size (8-Bits)	Operand Size (32-Bits)

Table 9.1: Code Bits

Of course many times :) Now, we can say that in 32-Bit programs (most of the time!) :

- In the case of bye ptr[]: The Bit (w) == 0
- In the case of word ptr[]: The Bit (w) == 1 and there's an Operand-Size Prefix (66h)
- In the case of dword ptr[]: The Bit (w) == 1 and there's no Operand-Size Prefix.

You should go now practice some more opcodes and also you should take a look at the small utility included with this chapter.

9.1 Playing with [CODE] Utility

Let's see if we understood the previous chapter or not. Open the utility (CODE II) that's included with the chapter.

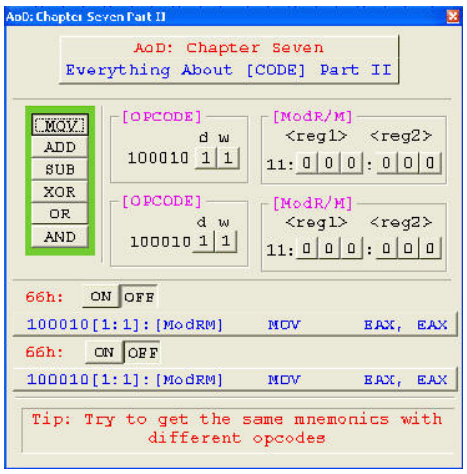


Figure 9.1: utility (CODE II)

Let's practice on an instruction from the above (MOV/ADD/SUB/..), let's say 'MOV'

```
MOV reg, reg
```

Let's change the ModRM Byte For the First Instruction to this:
The (111b) represents the register EDI in the ModRM Byte. Take a look at this table:

Figure 9.2: ModRM For First Instruction

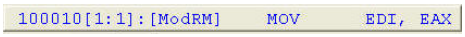


Figure 9.3: First Instruction Decoding

Let's leave the First Instruction for now, and go play with the second instruction. Let's change the ModRM Byte to this:
Obviously you can see the difference. In the first instruction, [edi] is the destination register while [eax] is the source. E.g. 'MOV' Instruction moves data from eax to edi. In the second instruction we have the opposite thing. In the second instruction, let's change the Bit (d) (Direction Bit) and set it to 0 instead of 1: Can you guess what will happen??

w=1		**w=0**
Reg	Value	Reg
EAX	000	AL
ECX	001	CL
EDX	010	DL
EBX	011	BL
ESP	100	AH
EBP	101	CH
ESI	110	DH
EDI	111	BH

Table 9.2: Values of the registers in ModRM Byte according to the Bit (w)

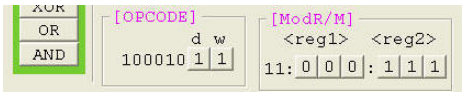


Figure 9.4: ModRM For Second Instruction

Figure 9.5: Second Instruction Decoding

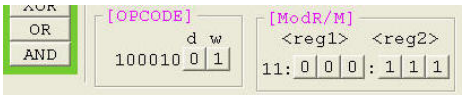


Figure 9.6: Second Instruction Decoding

Figure 9.7:

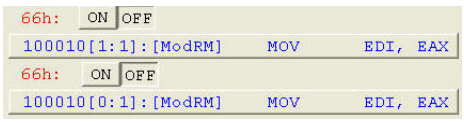


Figure 9.8:

Yes, the same mnemonics with different opcodes. In Hex (I decided not to show the hex value so you can go make it manually and practice more) it would be:

```
First Instruction:     8BF8      mov         edi, eax
Second Instruction:    89C7      mov         edi, eax
```

Now, it's time for you to go and practice! Please don't move further if you don't fully understand this chapter, if you have any questions after you read this chapter & followed the example, feel free to ask what you want.

9.2 Final Words

I guess we've learned the basics about [CODE] byte, things are getting more fun.

9.3 In the next chapter

We will read:

- Every Thing About ModRM Byte

Chapter 10

Everything about ModRM

We saw together in the early chapters of this tutorial, the basics of ModRM byte, in this chapter I hope we'll learn together everything it, and how to decode it. Let's quickly remember the basics of the structure of the ModRM Byte.

ModRM Byte, is decoded in a special way, in binary it's something like this: xx : xxx : xxx . 8 Bits are divided into three groups (2:3:3). Most of the time, the bits -as we said- are divided in three groups. Let us understand each of them.

XX : xxx : xxx (Mod) Bits :

- 00: memory address (e.g. eax, [eax])
- 01: memory address with 1 byte displacement (e.g. [eax+00]) (1-Byte [DISPLACEMENT])
- 10: memory address with 1 dword displacement (e.g. [eax+00000000]) (4-Byte [DISPLACEMENT])
- 11: both operands are memory (e.g. [eax, eax]

Let's understand the above four forms, and look at the following examples.

```
00401000   8B:C0           [00:000:000]  mov eax, eax
00401002   8B:00           [01:000:000]  mov eax, dword ptr ds:[eax]
00401004   8B:40:01        [10:000:000]  mov eax, dword ptr ds:[eax+00]
00401007   8B:80:00000001  [11:000:000]  mov eax, dword ptr ds:[eax+0000000]
```

We can easily see that the [OPCODE] byte is not changed (8B) in each instruction, but the following byte (ModRM Byte) changes. And corresponding to its encoding, we decide if there's additional bytes (DISPLACEMENT) after the ModRM byte. As In the third instruction, we have 1-byte displacement, while in the last (fourth) instruction we have a 1-dword displacement. Let's look at the same four instructions above after we change the bit (d) in the [OPCODE] byte as we practiced on the previous chapter.

```
00401000   89:C0           [11:000:000] mov     eax, eax
00401002   89:00           [01:000:000] mov     dword ptr ds:[eax], eax
00401004   89:40:10        [10:000:000] mov     dword ptr ds:[eax+00], eax
00401007   89:80:00100000  [00:000:000] mov     dword ptr ds:[eax+00000000], eax
```

We have the same four addressing modes but of course the direction is reversed (bit d). So that was about 32-Bit size. Is is the same for 16-Bit size? Do we still have the same four addressing mode?? Let's see what will happen for the same example if we were working with 16-bit operand size. We'll use the same four instructions with the Operand Size Prefix (66h) at the start of each of tem.

```
00401000   66:8B:C0          [11:000:000] mov     ax, ax
00401002   66:8B:00          [01:000:000] mov     ax, word ptr ds:[eax]
00401004   66:8B:40:01       [10:000:000] mov     ax, word ptr ds:[eax+00]
00401007   66:8B:80:00000001 [00:000:000] mov     ax, word ptr ds:[eax+0000000]
```

So far we talked about the four addressing modes in the ModRM byte for the 32-Bit & 16-Bit operand size, What about 8-bit size? Well, it's quite different, we'll see that later on this chapter.

xx : XXX : xxx Code/Reg field Bits :
The middle 3-bits in the ModRM Byte are the Code/Reg field bits, they can be decoded n treated like one of two things, Code Or Reg field. The processor knows which is the right decoding for this field (Whether it's Code or Reg field) from the [OPCODE] Byte itself! Let's see an example so, we don't get confused.

10.0.1 When considered as Code extension

We should know that there are instructions that require 1-Operand and others that require 2-Opernads. For example:

```
ADD:  instruction requires 2-Operands (add eax, eax)
SUB:  instruction requires 2-Operands (sub eax, eax)
```

While we have:

```
NOT:  instruction requires only 1-Operand (not eax)
MUL:  instruction requires only 1-Operand (mul eax)
DIV:  instruction requires only 1-Operand (div eax)
```

```
   11:010:000
   11:100:000
   11:110:000
```

If we're working with an instruction that requires only 1-opernad (as MUL instruction) then, the middle 3-Bits in the ModRM are 3-Code extension bits. Like (NOT/MUL/DIV) instructions, they all have 0xF7 as the [OPCODE] byte, while the difference is in ModRM byte.

```
00401009      F7D0      not     eax
0040100B      F7E0      mul     eax
0040100D      F7F0      div     eax
```

They have the same [OPCODE] the difference is only in the Code/Reg 3-Bits in the ModRM Byte, which in the previous instructions, are considered Code extension and the lowest 3-Bits in ModRM byte are the operand that the instruction requires.

10.0.2 When considered as Reg field

In other instructions that require 2-opreands, the middle 3-bits in the ModRM byte are considered as a reg value, the values are known to us, as we talked about them before, you can look at the table below. Now let's see an example.

```
00401025   33:83 78563412  xor eax, dword ptr ds:[ebx+12345678]
0040102B   33:8B 78563412  xor ecx, dword ptr ds:[ebx+12345678]
00401031   33:93 78563412  xor edx, dword ptr ds:[ebx+12345678]
```

Let's see the decoding of the ModRM Byte.

```
83:     [10:000:011]     10: == Addressing Mode | 000: == reg field (eax)
8B:     [10:001:011]     10: == Addressing Mode | 001: == reg field (ecx)
93:     [10:010:011]     10: == Addressing Mode | 010: == reg field (edx)
```

You may have noticed that 011 is the value for ebx.

xx : xxx : XXX reg/mem field Bits

Depending on the (Mode) Bits in the ModRM byte: In case of
`(11) this field means --> registers` we saw an example for this:

```
00401000   89:C0   [11:000:000]    mov  eax, eax
```

In case of (00, 01, 10):

- registers are pointers to memory
- There is a 'flag' that memory operand is specified by SIB byte
- There is a 'flag' that memory operand is specified by direct value

Mode == 00 :
If the reg/mem value == 101b & the mod == 00b then no register is used to calculate address, instead the address is in the DWORD after ModRM Byte.

Example:

```
00401023  8B:05:67452301   mov  eax, dword ptr ds:[1234567]
```

Note that 101b here, doesn't not stand for (EBP). It's a flag as we said that no register will be used to calculate the address and that this address is the DWORD right after the ModRM Byte. It means, we can't use (mov reg, dword ptr [ebp] with only 1-byte opcode & 1-Byte ModRM), anyway. If the reg/mem value == 100b & the mod == 00b then there's a SIB byte right after the ModRM byte. And again, 100b here, doesn't not mean (esp) it's a flag to indicate that there's a SIB byte.

Mode == 01 :
If the reg/mem value == 100b & the mod == 01b then there's a SIB byte right after the ModRM byte. And again, 100b here, doesn't not mean (esp) it's a flag to indicate that there's a SIB byte.

Mode == 10 : If the reg/mem value == 100b & the mod == 10b then there's a SIB byte right after the ModRM byte. And again, 100b here, doesn't not mean (esp) it's a flag to indicate that there's a SIB byte.

10.0.3 More Info About ModRM

First, we should know this:

- ModRM (and SIB as we'll see later) is used to specify operands.
- It may happen that an instruction have a ModRM byte
- It may happen that an instruction have a ModRM byte + SIB Byte (next chapter)
- It may NOT happen that an instruction have a SIB Byte without a ModRM byte (next chapter)
- We can say that SIB Byte is an extension of the ModRM Byte (next chapter)
- The format of ModRM Byte (and also SIB Byte) is 2:3:3

10.1 Playing With Our Tool

Open the tool attached with this chapter (ModRM 32) . (I'm using here the XP Version)

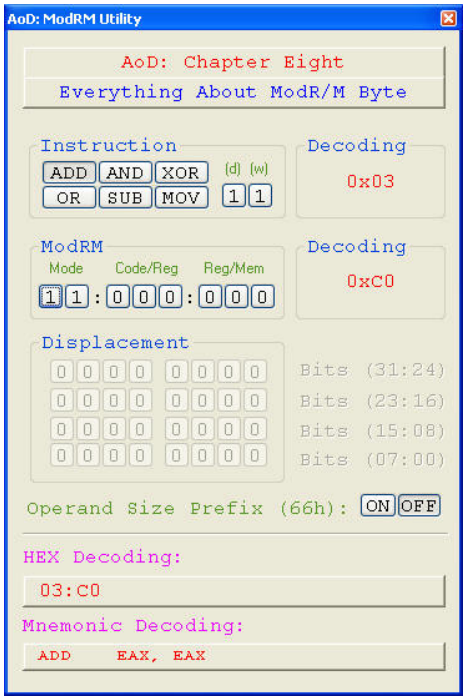


Figure 10.1: (ModRM Demonstrating Tool) -XP Version

Layout & Basic Overview Of The Utility:

You can find the default decoding options are like the following:

```
Default Instruction                    =     ADD
Default State For Bit (d)              =     1
Default State For Bit (w)              =     1
Default Mode For ModRM Byte            =     11b
Default reg1 Code For ModRM Byte       =     000b
Default reg2 Code For ModRM Byte       =     000b
```

Figure 10.2:

- Instruction: With these buttons, you can change the instruction to be decoded.
- Bit (d): With this button, you can change the default state for Bit (d) in the [CODE] byte.
- Bit (w): With this button, you can change the default state for Bit (w) in the [CODE] byte.
- Decoding: This one display the hex value for the [CODE] byte, changes depending on the above three options.

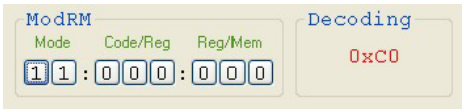


Figure 10.3:

- Mode: With these two buttons (2 Bits) you can change the default (11b) decoding more for ModRM Byte.
- Code/Reg: With these three buttons (3 Bits) you can change the default code for the Code/Reg.
- Reg/Mem: With these three buttons (3 Bits) you can change the default code for the Reg/Mem.
- Decoding: This one display the hex value for the [ModRM] byte, changes depending on the above three options.

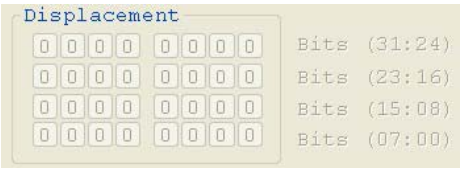


Figure 10.4:

Figure 10.5:

This part is disabled in the default mode for ModRM (11b), but it's enabled in certain ModRM modes, as we'll see shortly. The point is to manually write the displacement (whether a DWORD or a BYTE) Bit by Bit.

The next one makes you able to set if the Operand-Size Prefix (0x66) is enabled or not. This option makes you able to switch from 32-bit to 16-bit modes.

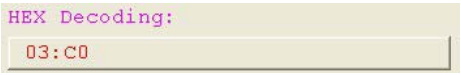


Figure 10.6:

This part as the name says, show the HEX decoding for the whole instruction. The main part (well, at least it's an important part). It shows the decoding for the instruction.

10.2 View Some Examples For ModRM Byte

Default Example (03:C0)

Let's look at the above example but of course as we used to look at it. Let's just ignore the first byte (the [CODE] byte) and just focus on the second byte (ModRM Byte). 0xC0 is the ModRM byte. Let's see it from the ModRM byte decoding style.

```
11 : 000 : 000 b
```

Figure 10.7:

Highest two bits (11) are -as we should know- the Mode bits. 11b means, both operands are registers. (e.g. [CODE] eax, eax). From decoding the Mode Bits, we know that the following 6 bits (which are separated into two groups, each consists of three bits), are two codes for registers, any of the 24 registers (8 registers * 3 Operand Sizes = 24). But how do we know which operand size are we working with? Is it 32-Bit register? Maybe 16-Bit? Maybe it's just 8-Bit register?? Well, we should easily tell the answer, as we talked about this in previous chapters. First we check if the Bit (w) in the [CODE] byte is zero. If it is, then we're in 8-bit size mode. If it's not, then we're whether in the 16-bit or the 32-bit modes. To know which one of them, we check for the operand-size prefix (66h) if it does exist, then we're in the 16-bit mode, if it does NOT exist, then we're in the 32-bit mode. Next is a table showing the 24 registers, for the three size modes.

(32-Bit)		(16-Bit)	(8-Bit)
Reg	value	Reg	Reg
EAX	000	AX	AL
ECX	001	CX	CL
EDX	010	DX	DL
EBX	011	BX	BL
ESP	100	SP	AH
EBP	101	BP	CH
ESI	110	SI	DH
EDI	111	DI	BH

(24 registers/ 8 registers for each of the 3 size modes)

The [CODE] Byte (03) when decoded in binary format : (00000 : 11) as we can see, Bit (w) is set, so we're dealing with full operand size (32/16 Bit Registers) and as there's no operand size prefix (66h) before the [CODE] byte, we know that we're dealing with 32-Bit operand sizes. Now back to the ModRM Byte (0xC0). The next three bits (Code/Reg Bits: 000) Are the register code to be used. 000b (like you can see in the table above) is the code for eax. Next we get the least significant three bits (Reg/Mem Bits: 000) which are also the code for eax. As we can see by looking at the decoding of the [CODE] byte, direction bit (Bit (d) == 1) which means the destination is Code/Reg and source is Reg/Mem.

So the final decoding will be: `add eax, eax`.

- We should check if a prefix (e.g. Operand Size Prefix (66h) is present
- We should know the decoding format for the current instruction we're decoding (in this example 00000:dw)
- We should extract both, Bit (d) & Bit (w)
- We should decode the following ModRM byte if exists (in the previous case, it exists)
- We should extract the Mode Bits, to know the correct decoding for the ModRM Byte
- We should decode the ModRM byte, depending on what ModRM mode we're dealing with

That's what should be done for decoding the above example. Now, let's take another example, for another Mode for the ModRM byte.

Another Example (66:8B:97:21436587)

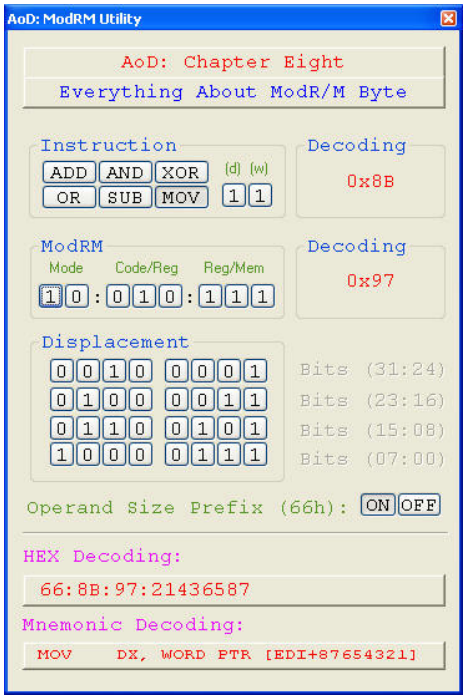


Figure 10.8: Example Two : Decoding The ModRM Byte

Now, let's look at a more -advanced- example. Let's see how can the ModRM byte be decoded for the previous example.

Following the simple steps we've seen in the previous example, the first thing we should do is checking if a prefix does exist. By looking at the first byte which is (66h) and looping through the prefixes table, we could easily know that this prefix is the Operand-Size prefix. That's nice, we know that we won't be dealing with 32-Bit operand sizes. It's 16-bit operand size, or 8-Bit operand size in case the Bit (w) is not set (in this case the prefix will be ignored, but this is not the case in our example, anyway).

As we said, a prefix (Operand-Size) does exists, we save this in memory for now, and let's move to the next step. Let's look at the [CODE] byte, in this example it's 0x8A the decoding for this instruction is the same as the previous example 00000 : dw , let's look at the byte in the binary format.

```
0x8A:     100010 : 11
```

The Five Most Significant Bits (100010b) are the code for the instruction 'MOV', next bit is the Bit (d), in this case it's set, which mean, Code/Reg is the destination register Reg/Mem is the source register. The least significant bit is the Bit (w), we see that it is set, so we know that we're working with full-size operand size (32-Bit/16-Bit) but as we remember we found an operand-size prefix, so we can say that we'll be working with 16-Bit operand size. Next step is to decode the ModRM byte (0x97), let's look at this byte in the correct format for a ModRM byte.

```
0x97 --> 10:010:111
```

We can see that the Mode is (10b) which means that registers here, are pointers to memory, no direct registers access, only memory access. As we saw in the case of (10b) as the Mode for a ModRM byte, we should check if the Reg/Mem value is (100b). If it is, then we know there's is a following SIB, but that's not the case here, as we're not dealing with SIB yet. By checking the Reg/Mem value, we see that it's (111b) so we know that there's no following SIB, instead we know that there is following DWORD displacement. By Decoding the Code/Reg 3-Bits, we see that it's (010b) which is the code for DX (remember we're working with 16-Bit size registers), and then, we decode the Reg/Mem 3-Bits which are (111b) which is the code for (edi), you can ask why it's not only (di), the answer is, edi here is a pointer to memory address, so it's 32-Bit size. Last thing to do is to decode the next DWORD, which is the displacement that should be decoded for a ModRM byte with the mode (10). We can see that the next dword is (0x21436587) which should be decoded as 87654321h. So, we by now decoded the previous instruction. The decoding is like this:

```
MOV DX,  WORD PTR [EDI+87654321]
```

Well, I hope this chapter with the two examples explained the ModRM byte to you, of course practicing on your own is the best way to fully-understand the ModRM Byte, so try playing with the ModRM byte utility that's attached with this chapter and also play with OllyDbg and try decoding several ModRM bytes on your own.

10.3 Final Words

Umm.. We're getting closer to building our disassembler, so I hope you understand everything we discussed in the previous chapters.

10.4 In the next chapter

We will read:

- Everything about SIB Byte

Chapter 11

Everything about SIB

11.1 What Does SIB Stand For?

SIB (SS : III : BBB) Stands for (Scale : Index :Base). General Format Of The SIB Byte (Base + Index * Scale). SS: Two most significant bits, are the code for the (Scale) (which can be considered as the multiplier) of the index register.

```
00:    = 2^0 = 1
01:    = 2^1 = 2
10:    = 2^2 = 4
11:    = 2^3 = 8
```

Some examples for the Scale of an SIB byte:

```
00 : *** : ***          ( mov reg, [reg*1] )
01 : *** : ***          ( mov reg, [reg*2] )
10 : *** : ***          ( mov reg, [reg*4] )
11 : *** : ***          ( mov reg, [reg*8] )
```

III: Next three bits, are the the (Index register) bits. They can contain any code for an index register, except the (esp register) we'll know why later. Some examples for the Index of an SIB byte:

```
00 : 000 : ***          ( mov reg, [eax*1] )
01 : 001 : ***          ( mov reg, [ecx*2] )
10 : 010 : ***          ( mov reg, [edx*4] )
11 : 011 : ***          ( mov reg, [ebx*8] )
```

BBB: The least three significant bits, are the (Base Register) code bits. Like the III bits. Some example for the Base of an SIB byte:

```
00 : 000 : 001          ( mov reg, [ecx + eax*1] )
01 : 001 : 010          ( mov reg, [edx + ecx*2] )
10 : 010 : 011          ( mov reg, [ebx + edx*4] )
11 : 011 : 000          ( mov reg, [eax + ebx*8] )
```

Now let's know why we can't use (esp) as the Index Register. If you couldn't guess it your self, the reason is quite similar to the exception we made in the ModRM Byte, simply the code for the register (esp) is used as a flag. Read next:

- If Index register code is the code for (esp), then the index is IGNORED, and in this case, the value of the scale is also IGNORED, and only the Base field is used to calculate the address.
- If we need to encode an instruction like (add reg, [esp]) it can't be done with simply using an SIB byte. But it would be encoded like this: (add reg, [esp + DISPLACEMENT])
  - If ModRM Byte (mode) == 01b then the displacement is 1-Byte
  - If ModRM Byte (mode) == 10b then the displacement is 1-Dword

11.2 Playing With Our Tool

Now, let's see some examples, on how to decode the SIB byte. General Layout Of The SIB Tool:

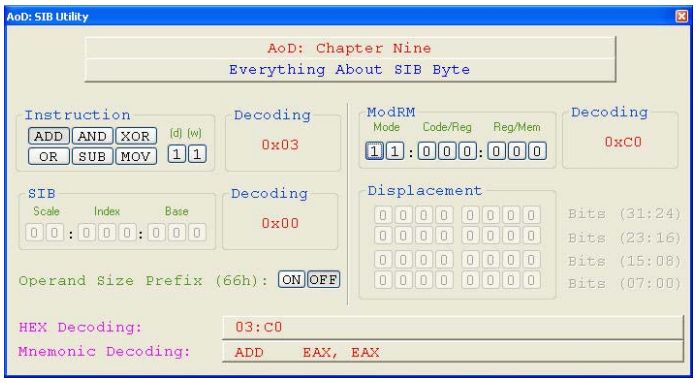


Figure 11.1: XP Version Of The SIB Tool

The layout is very similar to the ModRM Tool that we've played with in the previous chapter, the only difference is the new SIB Group Box, so let's get more familiar with it.

Figure 11.2: SIB Group Box

(SIB Group Box)
The new SIB Group Box, enables us to encode any SIB byte that we want. It's divided into four main groups:

```
SCALE:    The Scale (SS) Of The SIB Byte
INDEX:    The Index Register (III)
BASE:     The Base Register (BBB)
Decoding: Shows The HEX Value Of The SIB Byte
```

An Example. Let's say we want to decode this:

```
0040121C  66:2B84F9 BC6A3C89  sub ax, word ptr [ecx+edi*8+893C6ABC]
```

1. `66:2B84F9 BC6A3C89`
   First, we check for any prefix, in the previous case we find the Operand-Size Prefix (66h), so we know we're not working with 32-Bit operand size, it's whether 8-Bit or 16-Bit.

2. `66:2B84F9 BC6A3C89`
   0x2B is the [CODE] for the instruction 'SUB' and it requires a ModRM byte, so we know the following byte is a ModRM Byte. Also by decoding this byte in the [CODE] format: (001010:11)

   - Bit (d) == 1 : Now we know the direction of the instruction flow (e.g. we know the Source/Destination)
   - Bit (w) == 1: So, we know we're in 16-Bit size mode.

3. `66:2B84F9 BC6A3C89`
   0x84 as we knew, is the ModRM Byte, so let's decode it like a ModRM Byte. 10 : 000 : 100b

   - 10: Is The Mode, it means there will be a 32-BIT (DWORD) Displacement.
     The displacement follows the ModRM Byte if there's no SIB byte, else, it follows the SIB Byte.
   - 000: Is The Code/Reg code, here, it's the code for the register (AX) (remember we're working with 16-Bit Operand Size)
   - 100: Is NOT the code for (esp). It is a special flag telling the processor that there will be another following SIB byte, so we know that the following byte is an SIB byte.
   - So The ModRM Byte Decoding Will Be (AX, WORD PTR [SIB+ DISP32])

4. `66:2B84F9 BC6A3C89`
   0xF9 as we knew, is the SIB Byte, let's decode it as a SIB Byte: 11 : 111 : 001b

   - 11: The Scale (3d) == (2^3 == 8)
   - 111: The Index Register (EDI)
   - 001: The Base Register (ECX)
   - The decoding for the SIB Byte, would be (Base +Index*Scale) == (ECX+EDI*8)
   - The decoding for the ModRM Byte & SIB Byte will be (AX WORD PTR [ECX+EDI*8 + DISP32)

5. `66:2B84F9 BC6A3C89`
   0xBC6A3C89 Is the Displacement (we knew that from the ModRM Byte) it's decoded (little endian) like this (0x893C6ABC). So the complete decoding will be (SUB AX WORD PTR [ECX+EDI*8 + 893C6ABC)

I hope the example made you understand better the SIB byte decoding, the best way to fully understand it, is by practicing. Use the tool included + OllyDbg, and enjoy!

11.3 Final Words

So far, we've learned a lot about the Intel Instruction Format, we'll finish it in the next chapter, then we'll start coding the engine for out disassembler.

11.4 In the next chapter

- Everything About the Displacement

Chapter 12

Everything About Displacement

Displacement is sometimes required by some addressing forms, it comes right after any ModRM Byte or SIB Byte (if present). Displacement can be 1,2 or 4 bytes. Look at the following tables from the Intel Manuals.

Effective Address	Mod	R/M
[BX+SI]	00	000
[BX+DI]		001
[BP+SI]		010
[BP+DI]		011
[SI]		100
[DI]		101
disp16[2]		110
[BX]		111
[BX+SI]+disp8[3]	01	000
[BX+DI]+disp8		001
[BP+SI]+disp8		010
[BP+DI]+disp8		011
[SI]+disp8		100
[DI]+disp8		101
[BP]+disp8		110
[BX]+disp8		111
[BX+SI]+disp16	10	000
[BX+DI]+disp16		001
[BP+SI]+disp16		010
[BP+DI]+disp16		011
[SI]+disp16		100
[DI]+disp16		101
[BP]+disp16		110
[BX]+disp16		111

Effective Address	Mod	R/M
[EAX]	00	000
[ECX]		001
[EDX]		010
[EBX]		011
[–][--][1]		100
disp32[2]		101
[ESI]		110
[EDI]		111
[EAX]+disp8[3]	01	000
[ECX]+disp8		001
[EDX]+disp8		010
[EBX]+disp8		011
[–][--]+disp8		100
[EBP]+disp8		101
[ESI]+disp8		110
[EDI]+disp8		111
[EAX]+disp32	10	000
[ECX]+disp32		001
[EDX]+disp32		010
[EBX]+disp32		011
[–][--]+disp32		100
[EBP]+disp32		101
[ESI]+disp32		110
[EDI]+disp32		111

Figure 12.1: 16-Bit and 32-Bit Addressing Forms with the ModRM Byte

In the previous two tables, it shows when displacement is required, depending on the ModRM and the addressing mode (remember address-size prefix?). Let's see few examples:

1. 32-Bit Addressing Mode

   - `8B:05 00000010  [CODE][MODRM][DISP32] mov eax, dword ptr ds:[10000000]`
     `[CODE]   --> 'MOV'`
     `[MODRM]  --> 00:000:101 == [DISP32]`
     `[DISP32] --> 00000010 == 10000000`
   - `8B:40 10     [CODE][MODRM][DISP32] mov eax, dword ptr ds:[10]`
     `[CODE]   --> 'MOV'`
     `[MODRM]  --> 01:000:000 == [EAX+DISP8]`
     `[DISP8]  --> 10 == 10`
   - `8B:80 00000010  [CODE][MODRM][DISP32] mov eax, dword ptr ds:[eax+10000000]`
     `[CODE]   --> 'MOV'`
     `[MODRM]  --> 10:000:000 == [EAX+DISP32]`
     `[DISP32] --> 00000010 == 10000000`

2. 16-Bit Addressing Mode

   - `67:8B06 1234` `[PREFIX][CODE][MODRM][DISP16] mov eax, dword ptr ds:[3412]`
     ```
     [PREFIX] --> 16-Bit Addressing Mode
     [CODE]   --> 'MOV'
     [MODRM]  --> 00:000:110 == [DISP16]
     [DISP16] --> 1234 == 3412
     ```
   - `67:8B40 12` `[PREFIX][CODE][MODRM][DISP8] mov eax, dword ptr ds:[bx+si+12]`
     ```
     [PREFIX] --> 16-Bit Addressing Mode
     [CODE]   --> 'MOV'
     [MODRM]  --> 01:000:000 == [DISP8]
     [DISP8] -->  12 == 12
     ```
   - `67:8B06 1234` `[PREFIX][CODE][MODRM][DISP16] mov eax, dword ptr ds:[bx+si+3412]`
     ```
     [PREFIX] --> 16-Bit Addressing Mode
     [CODE]   --> 'MOV'
     [MODRM]  --> 10:000:000 == [DISP16]
     [DISP16] --> 1234 == 3412
     ```

Well that's it. I hope the previous examples (with previous chapters & previous two tables) gave you all you need to understand what displacement is. Go and practice now!

12.1 Final Words

This chapter was really short (The next chapter will be shorter!). There wasn't much to mention as we've already talked (maybe indirectly) about displacement & saw how they work in the previous two chapters (ModRM & SIB).

12.2 In the next chapter

- Everything About Immediates

Chapter 13

Everything About Immediates

Basically, immediate operand, is a value which is used (as-is) in the opcode. This value can NOT be changed as a result from a previous instruction. This value is NOT a memory address. This value is -if we can say- a constant value which the instruction may use. To explain this more, read next. Please have a look at the following instructions:

```
ADC:

0x14  ADC AL,imm8      ' Add with carry
0x15  ADC EAX,imm32    ' Add with carry

Example:
00401000  14 10        adc  al,  10    (0x14:      000101:0:0)
00401002  15 00000010  adc  eax, 10000000 (0x15:      000101:0:1)

ADD:

0x04        ADD AL,imm8             ' Add
0x05        ADD EAX,imm32           ' Add
00401000   04 10          add     al, 10       (0x04: 000001:0:0)
00401002   05 00000010    add     eax, 10000000 (0x05: 000001:0:1)
```

Looking the LSB, we can say (as we knew in the previous chapters) that this bit is the (w) bit, which is responsible for the operand size (is it Full-Size (32-Bit/16-Bit)or Partial Size (8-Bit). But what about the next bit? We saw this bit (Bit[1]) in the previous chapters -for some 1-byte opcodes- as the Bit (d) or the bit which is responsible for telling what operand is the source and what is the destination. But if you had a look at the previous examples above, you can see that we are dealing with an (immediate) value. Each instruction above refers to an immediate value. It's not a memory address, just a value. For example (add eax, [10000000]) is not the same as (add eax, 10000000), the first is adding the value in a memory address, the second as adding the value itself. I think it's pretty obvious. so, back to that bit (Bit[1]). When we're dealing with an instruction that uses an immediate value such as the example above, do we expect to have a bit (d) that's responsible for the direction of Source/Destination ? Could we ever see something like:

```
(00401000  add     10000000, eax)
```

Is such an instruction valid? Of course NO! We can NOT have an immediate value as the destination operand. And thus, instructions dealing with immediate operands, does NOT need such a bit -Bit (d)- as the immediate operand is ALWAYS the source operand. Above examples were meant to make us understand what the immediates are, and why there can't be a bit (d) in an instruction using an immediate operand. So, forget about the previous examples for now, and go on.

13.1 Bit (s) : A New Special Bit

So is the bit [1] in an instruction using an immediate operand does have a special name/meaning? Yes, it does. The bit[1] from now on will be referenced as Bit

(S). Let's understand what does it mean. The description of the Bit (S) from the Intel Manuals:

> Sign Extend (s) Bit: The sign-extend (s) bit occurs primarily in instructions with immediate data fields that are being extended from 8 bits to 16 or 32 bits. Table B-5 shows the encoding of the s bit.

```
Bit (S)    Effect on 8-Bit Immediate Data    Effect on 16- or 32-Bit Immediate Data
   0                    None None
   1              Sign-Extend to fill 16-Bit            None
                  or 32-Bit destination
```

From the previous table, we can say that Bit (S) has no effect on a (16/32-Bit) immediate operand. It only affects 8-Bit immediate operands. Well, not clear? Let's see the following example:

```
80D0 11            adc al,11 (100000:0:0 Bit (s) == 0 Bit (w) == 0)
81D0 11223344      adc eax,44332211   (100000:0:1 Bit (s) == 0 Bit (w) == 1)
82D0 11            adc al,11 (100000:1:0 Bit (s) == 1 Bit (w) == 0)
83D0 11            adc eax,11   (100000:1:1 Bit (s) == 1 Bit (w) == 1)

80 s = 0 , w = 0 (immsize = full, operand = byte)
81 s = 0 , w = 1 (immsize = full, operand = full)
82 s = 1 , w = 0 (immsize = sbyte, operand = byte)
83 s = 1 , w = 1 (immsize = sbyte, operand = full)
```

If you don't understand this chapter, or you find any difficulty, it's OK. Just read it again, and then read next chapters, you'll get more used to it in next chapters.

13.2 Final Words

This chapter was the last part of the Intel Instruction Format. We should by now know everything (at least most of the knowledge) about the Intel Instruction Format. In the next chapter we'll cover some few things that we didn't mention, about the Intel Instruction Format. After the next chapter, we'll go into an amazing journey, making our disassembler. We'll have a great time coding our disassembler, and watching how it really decodes the bytes :) But as a suggestion from me, please don't go on before you understand the previous chapters. If you do have some questions that the previous chapters couldn't answer, you know where to ask us questions.

13.3 In the next chapter

- Final Words About The Intel Instruction Format

Chapter 14

Final Words About The Intel Instruction Format

14.1 ModRM 16-Bit

In the previous chapter we saw how to decode ModRM byte (and included a tool) but only 32-Bit version. We didn't talk about 16-Bit mode. In this chapter we'll see how to decode 16-Bit ModRM Byte. Let's look at the following table extracted from the Intel Manuals.

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX+DI]+disp8		001	41	49	51	59	61	69	71	79
[BP+SI]+disp8		010	42	4A	52	5A	62	6A	72	7A
[BP+DI]+disp8		011	43	4B	53	5B	63	6B	73	7B
[SI]+disp8		100	44	4C	54	5C	64	6C	74	7C
[DI]+disp8		101	45	4D	55	5D	65	6D	75	7D
[BP]+disp8		110	46	4E	56	5E	66	6E	76	7E
[BX]+disp8		111	47	4F	57	5F	67	6F	77	7F
[BX+SI]+disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI]+disp16		001	81	89	91	99	A1	A9	B1	B9
[BP+SI]+disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI]+disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI]+disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI]+disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP]+disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX]+disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	EQ	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Figure 14.1: 16-Bit Addressing Forms with the ModR/M Byte

Decoding ModRM Byte in 16-Bit Addressing Mode, can be considered easier than 32-Bit Mode, as there're not much exceptions in decoding. You should (as you're reading chapter twelve) have knowledge for understanding what does the previous table mean, and how to decode a ModRM byte in 16-Bit address mode, by looking at the table only! Attached with this chapter is a tool, to help you practicing with the decoding.

14.2 Final Words

In this short chapter, we -almost- finished everything in the Intel Instruction Format. It's impossible not no forget something, but you can make sure that if we did forget something it'll be mentioned later in the real engine. Starting from next chapter, the fun begins :)

14.3 In the next chapter

- Building The Decoding Engine Skeleton

Chapter 15

Building The Decoding Engine Skeleton

15.1 Before Starting

Before starting I must say that the project is based on the (AoD Beta) project we made in chapter four. I found it important to mention, so you don't get confused.

15.2 Constructing A Bytes-Parser

A Bytes-Parser -if we can call it- is the procedure that will loop through the bytes we need to disassembler. Let's understand the following procedure clearly before we construct our Byte-Parser.

15.2.1 Understanding Of Old Code

Inside (MenuHandling.inc) : MnuFileOpen

The procedure starts by invoking the (GetOpenFileName) API, which is responsible for showing the 'Open File' Dialog.

```
;------------------------------------ Open file -------------------
        ; Display the "Open File" dialog box
;------------------------------------------------------------------

        invoke GetOpenFileName, ADDR ofn
```

The return value is (TRUE) if we selected a file to open (e.g. didn't click 'cancel'), else the return value is (FALSE). We need to check the return value so we can decide, should we continue or not.

```
;------------------------------------------------------------------
        .if eax==TRUE  ; Didn't click CANCEL
;------------------------------------------------------------------
```

After this, we can continue normally to load the file, map it to memory. After successfully loading the file, we call the procedure (CheckSignatures) to check whether the file is a valid PE file. If the file is a valid PE file, we continue by calling the procedures (GetNumberOfSections - GetSections - GetEntryPoint). You should check the sources for full details.

Writing New Code:

So a Bytes-Parser will actually loop through the bytes of the code section. So, we need to write a procedure that will get the code section, and it's length so we can -parse- its bytes. We can do this, by looping through the sections of the loaded PE, and check them against the EntryPoint. Once a section that contains the EntryPoint is found, we know that this is the code section we should disassemble (We're not dealing with any kind of packers in the current stage of AoD). Let's write the (GetCodeSection) procedure.

Inside (PEStuff.inc) : GetCodeSection

```
mov eax, EntryPointRVA              ; RVA Of The EntryPoint
invoke RVAToOffset,eax              ; ebx == Section Index (Code Section)
mov CodeSectionIndex, ebx           ;
mov eax, ebx                        ;
mov ecx, sizeof SECTION             ;
mul ecx                             ;
lea esi, [Sections]+eax             ; esi = Sections[CodeSectionIndex]
assume esi: ptr SECTION             ;
mov eax, [esi].VirtualAddress       ;
mov CodeStartRVA, eax               ; Save Code Section Start (RVA)
mov eax, [esi].PointerToRawData     ;
mov CodeStartOffset, eax            ; And Offset.
mov eax, [esi].SizeOfRawData        ;
mov CodeSize, eax                   ; Save Code Section Size
ret
```

This procedure, uses the (RVAToOffset) to get the Code Section Index, and then get its offset & its size. The offset & size are saved to (CodeStartRVA / CodeStartOffset & CodeSize). We'll need these values later in our parser. So now, we have the start and the length of the code section. Let's write a simple (fake) parser. (What's the use of such -fake- parser? Well, to simulate the process of disassembling the code section, so the fake parser will just display all the bytes in the code section only)

Inside (BytesParser.inc) : ParseCodeSection

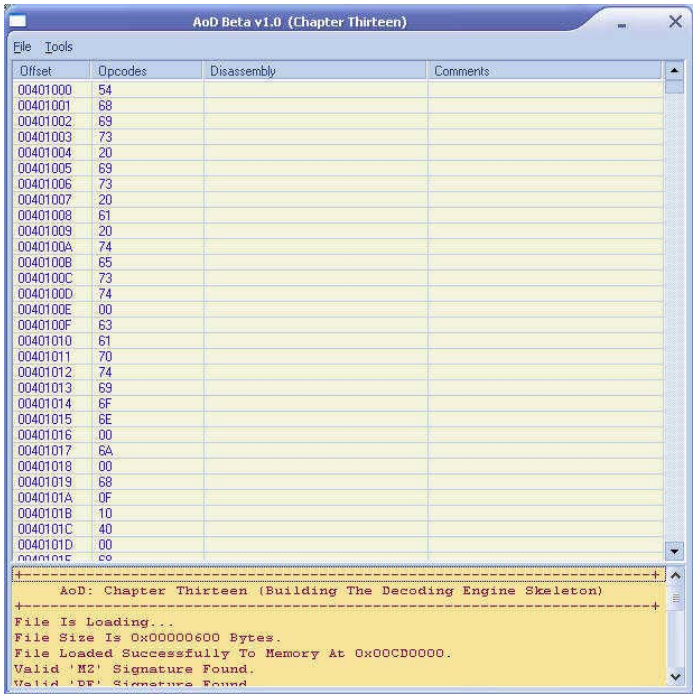
```
ParseCodeSection    proc
    invoke ShowWindow, hDsmList, FALSE
    mov esi, FileOffset
    add esi, CodeStartOffset
    xor ecx, ecx
    xor eax, eax
    mov edx, CodeSize
    .while ecx < edx
        mov al, byte ptr [esi]
        pusha
        invoke wsprintf, addr szTemp2, addr szFormat, eax
        popa
        pusha
        mov eax, 00400000h
        add eax, CodeStartRVA
        add eax, ecx
        invoke wsprintf, addr szTemp, addr szFormat2, eax
        popa
        invoke InsertItem, ecx, 0,addr szTemp
        invoke InsertItem, ecx, 1,addr szTemp2
        inc esi
        inc ecx
    .endw
    invoke ShowWindow, hDsmList, TRUE
    ret
ParseCodeSection endp
```

And Some Data For The Fake Parser

```
.data?
szTemp  db  250 dup (?)
szTemp2 db  250 dup (?)

.data
szFormat    db "%.2X",0
szFormat2   db "%.8X",0
```

As I said, the previous procedure does nothing more than simulating a disassembler parsing routine, and test our code that gets the code section offset & size. It's therefore not necessary to optimize it in anyway, as it'll be changed -as we'll see next-. The output from the previous procedure (for the msgbox.exe attached) would be like this:




```
00401000| 54
00401001| 68 69732069
00401006|v73 20
00401008| 61
00401009| 207465 73
0040100D|v74 00
0040100F| 6361 70
00401012|v74 69
00401014| 6F
00401015| 6E
00401016| 006A 00
00401019| 68 0F104000
0040101E| 68 00104000
00401023| 6A 00
00401025| E8 02000000
0040102A| C3
0040102B| CC
0040102C|-FF25 00204000
```

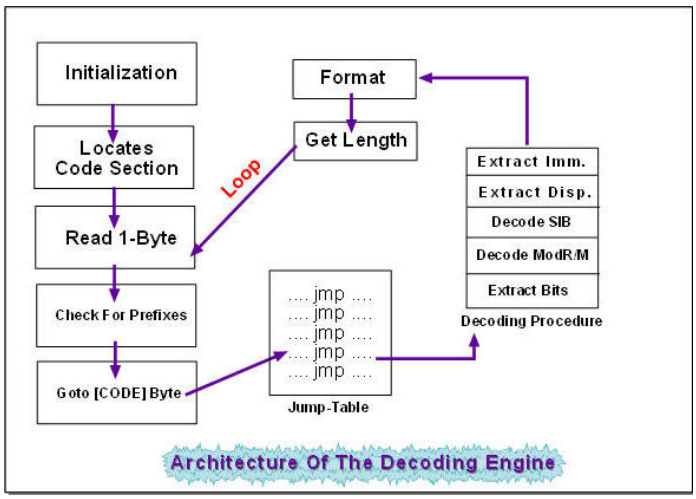
15.3 Idea Of A Real Engine Skeleton

Now instead of making a -fake- parser, let's do the real engine. Let's build a skeleton for the decoding engine. To accomplish this, let's put some points that should be included in a real engine:

- Engine Initialize : Set the necessary initialization data
- Locates the code section
- Read 1-Byte
  - Check for prefixes & save data for later use
  - Get the [CODE] byte
  - Use this byte to jump to the necessary procedure to decode this byte (using a jump-table)
    - The (jump-table) will -jump to the corresponding decoding procedure for each [CODE] byte
    - Each decoding procedure will use (all/some) of the following procedures
      - A procedure to extract bits from [CODE] byte. (e.g. Bit (d) / Bit (s) / Bit (w))
      - A procedure to decode a ModRM byte.
      - A procedure to decode an SIB byte. (We get this from the previous procedure).
      - A procedure to get the displacement/immediate value.
    - Invoke the formatting procedure, which will join the result of all previous procedures.
  - Calculate the length of the whole instruction, and set the index to the next instruction.
- Go to next byte to be decoded.
- Loop.

That was the layout of a real decoding engine. This is what we're going to build in the next chapter. Take a look at the picture for better understanding of the layout.

You should've got the idea now, on how our engine will work. This is the basic-layout, in next chapters, there'll be some changes, but the architecture remains the same.



15.4 Final Words

In this chapter we saw what is a Byte-Parser. We coded a fake one, just to see how to extract the bytes that need to be disassembled (code section) from a PE file. And the most important thing, we talked about the idea of a real decoding engine. In the next chapter, we'll design the decoding engine, step-by-step, based on the idea we talked about in this chapter. Obviously, it's getting more fun.

15.5 In the next chapter

We will read

- Designing The Engine

Bibliography

[1] Hutch. hutch's home page - masm32 download page. http://www.movsd.com/ , http://www.masm32.com/, 2003.

[2] Iczelion. Iczelion. 2003.

[3] KetilO. Radasm© win32 assembly ide for masm/tasm/fasm/nasm/goasm/hla. http://radasm.visualassembler.com/, 2003.

[4] Luevelsmeyer. Luevelsmeyer. 2003.

[5] MASM. Masm. 2003.

[6] Microsoft(C). Msdn developer centers. http://msdn.microsoft.com/, 2003.

[7] OllyDbg. Ollydbg. 2003.