The Art of Machine Language Programming

By Joel Jakubovic

Introduction

Many years ago, when I was first learning to program though BASIC and Batch files, I had a rather naïve dream. I knew that I could use BASCOM to turn those .bas files into .exe's, and later on when I learned C and C++ they produced executables as their end result. However when I opened them in Notepad they were a mess of random characters that looked nothing like the source files. In my quest to go further and further low-level, I learned x86 Assembly and assembled a few small tests, but I still couldn't make head nor tail of the resulting file formats.

My eventual dream was to be able to 'write' an EXE, in raw hex. No compilation, assembling or linking needed. However, no amount of Googling could give me anything on the matter, probably because I didn't know what to search for. Whenever I searched for “machine code tutorial” it came up with dated stuff for ancient computers, and not Windows. I had yet to learn it's not as simple as just one “machine code” for all processors. It was when I stumbled across **Iczelion's Win32 assembly website** (http://win32assembly.online.fr/tutorials.html) that everything clicked and began to make more sense. I had to grasp the basics of the C Win32 API first, but once I had made myself comfortable with that, and done a few Google searches with the correct keywords, I achieved my goal :)

The things that made this whole process harder and slower were probably my age when I first had the idea (around 9 or 10 years old!!), my lack of experience as I was still a n00b to programming for several years, and the fact that I was almost completely self-taught. Google has been my best friend throughout, but despite this it's really difficult to learn completely by yourself. One of my greatest mistakes was neglecting to just ask people on forums as a last resort – I think I felt I was cheating in some way by asking for specific help, but in retrospect I was pretty foolish not to.

Anyway, so the purpose of this text I'm writing is to explain, by example, how to write executable files in hex.

Why bother?

Exactly! Trust me, once you've done it you don't want to do it again, unless of course you want to write a tutorial on it. Assembly's the lowest level people tend to go for a reason. However it's still good to have an understanding of the formats, which are critical in reverse-engineering and other hacker-ish stuff like that. It's also a no-brainer that if you're working on a compiler / assembler of your own, you need to thoroughly understand the format of the end result :P

I personally thought it as a challenge. If I could understand this, which is the lowest software level I can think of, I'd be satisfied.

Prerequisites

Firstly I will assume a prior knowledge and understanding of at least one programming language, since this is a programming text after all. Since this text is on **x86 machine language**, you should also know basic x86 assembly, both 16-bit and 32-bit.
For 16-bit assembly, **A Brief x86 Assembler Tutorial** (http://www.skynet.ie/~darkstar/assembler/) is a great introduction. For 32-bit and 16-bit, I found **The Art of Assembly Language Programming** (http://maven.smith.edu/~thiebaut/ArtOfAssembly/artofasm.html) to be an incredibly detailed introduction to the subject (the title of this text is a nod to it :D). I also found the **x86 Disassembly** Wikibook (http://en.wikibooks.org/wiki/X86_Disassembly) helpful for understanding the function calling mechanism and the stack. A basic knowledge of the **Win32 API** and how it's used in Assembly is also needed, and **Iczelion's site** above is invaluable for this. The emphasis here is on 32-bit, Windows assembly but the 16-bit knowledge is still useful for understanding of some older file formats. An understanding of the **hexadecimal and binary** number systems (including two's complement representation) is also critical, in addition to basic memory concepts such as addresses, address space, and the like.

Overview

The first thing to know here is that there's no such thing as *the* ".exe" file format. Under Windows, this extension covers at least three different formats used for current versions of Windows and to maintain backwards compatibility with old programs.

When I first 'tried' to write an EXE by hand, I had no knowledge of assembly and only basic knowledge of programming in general. So my first attempt involved creating a blank text file and typing in "Hello, World!" in binary. I saved it as a .exe and was confused when my program didn't run at all, let alone display "Hello, World!" to standard output...!
Of course, this was because I had no knowledge of the file formats used. Later on, with knowledge of assembly, I thought .exe's consisted of machine instructions from start to finish. This isn't quite the case for .exe's, but there **is** another lesser-seen extension that is exactly this: **.COM**.

COM files are probably the simplest executable to understand. A relic from DOS, they are 16-bit, have a maximum size of 65536 bytes, and do not use segmentation. They're loaded at address `0x100` in DOS' memory, and execution begins right there, at the start of the program.

COM files will be the first to be covered in more detail because of their simplicity. However we also have .exe files to deal with. As mentioned, exe's cover three different 'sub-formats': **MZ**, **NE**, and **PE**.

An **MZ**, or 'DOS' executable file, starts with the letters "MZ" and is 16-bit. Good old BASCOM produced these. It is comprised of a 'header' at the beginning and then mainly instructions or data.

NE stands for New Executable, and is a really rare type that was used for early 16-bit versions of Windows. Unfortunately I don't understand this format fully and I don't really see it worth understanding, since so few programs use it (**Zeek the Geek FTW!**), so NE's will be skipped in this text. Some documentation is available on the internet for it.

Now, despite the three previous formats being for DOS or 16-bit Windows, 32-bit Windows maintains backwards compatibility with them and can still run them. This is why I could use ancient BASIC interpreters from DOS to help me learn BASIC, on Windows XP :)

Lastly, we have **PE**, or "Portable Executable". This is the format used by the current modern, 32-bit versions of Windows and is a bit more involved than the MZ executable. It consists of a mini MZ executable at the beginning, a mass of headers, a table, and then code/data.

So, without further ado, let's have a look at the first executable format in the list.

Chapter 1: COM files

To provide a starting point, suppose we have the typical “Hello World” assembly program to convert into a .COM file:

```
mov ah,09
mov dx,OFFSET msg
int 21h                    ;display message
mov ah,4C
mov al,00
int 21h                    ;exit
msg db “Hello, World!$”
```

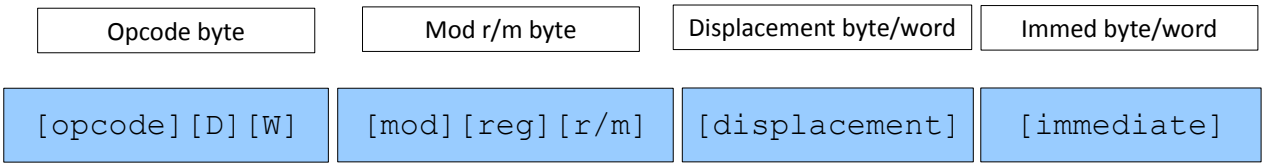
Since a COM file is just this but in numerical, instruction format, all one needs to do is convert.

ASM to machine code conversion

Unless you know all of the CPU's instructions by heart, you'll need a reference. I use MASM32's Opcodes list mainly but I also sometimes use http://ref.x86asm.net/coder32.html. I will refer to MASM's reference pages here but the above website uses similar notation.

The only things the processor can understand are numbers. These numbers, when interpreted and executed by the CPU as instructions, are known as machine code or machine language. Different processors have different machine languages, but fortunately Intel's are almost fully backward compatible with each other. An x86 machine instruction covers both the operation (eg `mov`) and the operand(s) (eg `ah` and `09`). Each assembly language instruction corresponds to at least one numerical machine instruction. I say 'at least one' because some instructions that look like the same operation (such as `push 4Ch` and `push ax`) actually have different numerical codes, as will be seen...

To be able to use these resources you need to know the general format for 16-bit instructions. This is:



The first box is called the Opcode byte. 'Opcode' is short for 'operation code' and gives a general idea of what the operation is (eg add, mov, push) and the type of operands it accepts. The `[opcode]` field is **6 bits** long and represents the actual operation to be performed. `[D]` is a **single bit** that specifies the **direction** of the operands; if it is `0`, the register represented by the `[reg]` field of the mod r/m byte is used as the **source**, and if it is `1`, `[reg]` is used as the **destination**. `[W]` is another single bit that determines the **width** of the operands; `0` means they are 8 bits (1 byte) wide and `1` means they are 16 bits (1 word / 2 bytes) wide.

The mod r/m byte determines the specific operands of the instruction. It's not always present in all instructions, particularly those that have an immediate operand (will be covered shortly).

The `[reg]` field represents a register as one of the operands (as mentioned, it can be either the source or destination depending on the `[D]` bit). It can be either 8-bit, if `[W]` is 0, or 16-bit if `[W]` is 1. These are the values with their corresponding registers:

```
[reg]       [W]=0       [W]=1
000           al          ax
001           cl          cx
010           dl          dx
011           bl          bx
100           ah          sp
101           ch          bp
110           dh          si
111           bh          di
```

`[mod]` ('mode') is a 2-bit field that can affect how the `[r/m]` field is treated. `[r/m]` is 3 bits and can represent either a register or a memory address. The effects of `[mod]` are:

`00` - If `[r/m]` is `110`, treat `[displacement]` as the address. If not, there is no `[displacement]`.
`01` - `[displacement]` is 8 bits, but sign-extend it to 16 bits
`10` - `[displacement]` is 16 bits
`11` - treat `[r/m]` as a second `[reg]` field

And for `[r/m]` treated as a memory address:

```
000 - bx + si + [displacement]
001 - bx + di + [displacement]
010 - bp + si + [displacement]
011 - bp + di + [displacement]
100 - si + [displacement]
101 - di + [displacement]
110 - bp + [displacement] unless [mod] = 00, as mentioned
111 - bx + [displacement]
```

A 'displacement' is, as can be seen, a (signed) value that is added to registers which 'displaces' it by some amount. This would be used in an instruction like `add ax,[bp + 2]`; in this case, `2` would be the displacement. Of course, displacements aren't always needed; this is when you'd use a `[mod]` of `00` to take it out of the picture. This would work for all registers except `bp` on its own, because of the exceptional case where the displacement is treated as the address. Instead for this you'd use a `[mod]` of `11` to treat `[r/m]` as a second register, and use `101` for this field (provided `[W]` is 1!).

Note that if `[mod]` is `11`, you can forget the above list for `[r/m]` and use the `[reg]` list instead. This enables register-register operations rather than just register-memory operations.

As a quick reference, the following table shows the combined effects of `[mod]` and `[r/m]` on the resulting operand:

[r/m]--> [mod] V	000	001	010	011	100	101	110	111
00	[bx+si]	[bx+di]	[bp+si]	[bp+di]	[si]	[di]	[disp16]	[bx]
01	[bx+si+disp8 (SX)]	[bx+di+disp8 (SX)]	[bp+si+disp8 (SX)]	[bp+di+disp8 (SX)]	[si+disp8 (SX)]	[di+disp8 (SX)]	[bp+disp8 (SX)]	[bx+disp8 (SX)]
10	[bx+si+disp16]	[bx+di+disp16]	[bp+si+disp16]	[bp+di+disp16]	[si+disp16]	[di+disp16]	[bp+disp16]	[bx+disp16]
11	ax	cx	dx	bx	sp	bp	si	di

An [immediate] is a value that is treated as-is rather than a memory address or a displacement. For example, in mov cl,03 the value 03 is an immediate. Most instructions taking immediates have custom opcode bytes and lack mod r/m bytes, since the operation is one-way (you can't switch the above instruction around since storing cl in 03 makes no sense).

Whew! I don't know whether that was hard to follow, but it was certainly hard to explain. If it's not crystal clear then hopefully the following examples will help. They will also discuss some of the conventions and notation used in instruction references.

Examples

1) mov cl,03

Let's start off with a really basic instruction: mov cl,03. The MASM page for mov lists quite a few different opcode bytes corresponding to different varieties of the instruction. Since this involves an 8-bit register and an immediate, look for the one that deals with an r8 (8-bit register) and an imm8 (8-bit immediate). It says:

"B0+ rb MOV r8,imm8 Move imm8 to r8"

B0 is the base opcode of the instruction. Since the source is an immediate value, there is no mod r/m byte. The + rb part means you add the [reg] code for the 8-bit destination register to the base opcode; the 'b' stands for Byte, denoting an 8-bit register only ('w' means 'word' = 16 bits). If we look up cl in the [reg] list, we find it is 010 (binary 2). So we add 2 to B0, giving B2. This is currently equivalent to: mov cl. All we need to do now is supply the 8-bit immediate 03. If you look at the 16-bit instruction format from before you'll see that the immediate comes after the displacement and the mod r/m byte. Since we don't use the latter two in this instruction, we just need to place it right after the opcode byte, giving a final result of:

mov cl,03 = B2 03

2) add ax,bx

Now for a slightly trickier one. This involves two registers so it's going to require a mod r/m byte. The documentation for the add instruction, for a r16 and a r/m16, looks like this:

"01 / r ADD r/m16,r16 Add r16 to r/m16"

Remember that we're dealing with two registers here, and because the [r/m] field can represent a register, this will **also** perform the **same** operation:

"03 / r ADD r16,r/m16 Add r/m16 to r16"

Note that the [D] and [W] bits are used in these opcode bytes! For both of them, [W] is 1, denoting 16-bit operands. In the first, [D] is 0 denoting [reg] as the source and [r/m] as the destination. In the second, [D] is 1 which denotes the opposite way round. So since [reg] and [r/m] can **both** represent registers, **either** of these instructions will do. So which one do you use? Whichever you prefer, really. This redundancy is unfortunately part of many instructions in the x86 architecture.

For simplicity we will choose the first one, 01. Since the [D] bit is 0, the [reg] field denotes the **source** register. Our source register is bx. Looking it up in the [reg] table gives 011 (binary 3). So that's the [reg] field sorted. Our destination register is ax, which has a code of 000. This goes in the [r/m] field, but to get this interpreted as ax and not something like [bx+si], the [mod] field has to be 11.

So, to summarise, [mod] = 11, [reg] = 011, and [r/m] = 000.
Thus the mod r/m byte is 11011000 in binary, or D8 in hex. Since we don't have a displacement or an

immediate, our final result is:

```
add ax,bx        =           01 D8
```

3) `push [0ABCDh]`

I have chosen this instruction because it makes use of three important things. Here is the documentation for the relevant `push` instruction, taking a `r/m16` operand (in this case it's the `m16` part as it's a memory address):

```
"FF /6 PUSH r/m16 Push r/m16"
```

What on earth does the `/6` mean? Well, consider this: this instruction takes a `r/m16` operand. This will require a mod r/m byte, and will take up the `[r/m]` and use the `[mod]` bits. However this leaves us with an unused `[reg]` field... since there's only one operand. So the designers decided to use the **same opcode byte** (`FF`) for **different instructions**, which are **diffrentiated** by the `[reg]` field of the mod r/m byte. The `/6` (I think it's called an 'opcode extension') denotes this value, so the `[reg]` field is `110`. If you used a different value for `[reg]` you'd probably wind up with a completely different operation; you can verify this for yourself if you look at the `FF` opcode:

```
"
FF CALL
FF DEC
FF INC
FF JMP
FF PUSH
"
```

These **all** share the same opcode, `FF`, and are differentiated solely by the opcode extension, aka their `[reg]` field. This can be seen in several other opcodes as well.

The second thing it makes use of is, obviously, memory addressing. So far we have our opcode and our `[reg]`. That leaves `[mod]` and `[r/m]`. The memory address is given as a 16-bit displacement (`0ABCDh`), so if you consult the mod r/m table from before you'll see that `[mod]` should be `00` and `[r/m]` should be `110`. So, our mod r/m byte is `00110110`, or `36` hex.

The displacement comes after the mod r/m byte, so all we need to do is append it, leaving a final code of `FF 36 AB CD`, right? **WRONG!** Well, more like not *quite* right. The only problem here is that the displacement bytes are actually *in the wrong order*! This is correct: `FF 36 CD AB`. The reason for this is that the x86 architecture is **little-endian**; this means that when a multi-byte numerical value (such as `ABCD`) is stored in memory, it is stored **least-significant byte** (LSB) first. `ABCD` is a two-byte value, so `CD` comes first and then `AB`. This is true for larger values too; later on, when we cover 32-bit stuff, values like `00402008` are actually stored as `08 20 40 00`.

So in summary, our final result:

```
push [0ABCDh]          =           FF 36 CD AB
```

Phew! We should be ready to translate that assembly program from before. About time too...

That assembly program from before

Here it is again:

```
mov ah,09
mov dx,OFFSET msg
int 21h                    ;display message
mov ah,4C
mov al,00
int 21h                    ;exit
msg db "Hello, World!$"
```

Let's tackle them one by one:

`mov ah,09:` base opcode: `B0`. It says `+r`, so `B0 + 4 (ah = binary 100, aka 4) = B4`
Append `09`: **`B4 09`**

`mov dx,OFFSET msg:` this is a bit tricky because we don't actually know what `OFFSET msg` is till we've done everything else! The most we can do at this point is to translate up to here. So the base opcode is `B8`, adding the register code `010` (2) for `dx` gives `BA`, and if we represent the currently unknown immediate with `XX XX`: **`BA XX XX`**

`int 21h:` this is really simple, just **`CD 21`**

`mov ah,4C:` here we can just use the code for the first instruction but with a different immediate:
`B4 4C`

`mov al,00:` base opcode = `B0`, adding `0` gives us... well, `B0`, and appending `00` gives **`B0 00`**

`int 21h:` same as before, **`CD 21`**

It is at this point that we have `msg` stored as ASCII. Up to this point, we have:

```
B4 09 BA XX XX CD 21 B4 – 4C B0 00 CD 21
```

If we count up the bytes taken up so far, we can see `msg` is at `0xD` bytes **from the start of the file**. We must remember that COM programs start at offset **`0x100`** in (DOS) memory, so the actual **address in memory** of `msg` is **`0x010D`**. So we can now replace the `XX XX` with `0D 01` (remember, little-endian) and add in `msg` itself:

```
B4 09 BA 0D 01 CD 21 B4 – 4C B0 00 CD 21 48 65 6C
6C 6F 2C 20 57 6F 72 6C - 64 21 24
```

Now, this is the hex code for the program – if you save it as a .COM file using a hex editor or similar (in MASM32 just go to Tools->Save hex file as binary) and run, you should see a nice "Hello, World!" program flash up for a millisecond. You could always add in a keystroke interrupt, or run it in Command Prompt.

There we go! You've (probably) just written your first program in raw, unadorned hexadecimal. And if you think *that* was long and drawn out, there's a lot worse to come :)

Chapter 2: 'MZ' .exe's

This brings us to our first 'sub-format' of the .exe extension. Unlike .COM files, MZ executables aren't quite as simple as instructions and data right from the beginning. Instead there is a 'header' at the beginning, containing information on various things like initial values of `CS:IP` and `SS:SP`. The header is needed to allow programs to take up **more than 1 segment** in memory, which was not the case in .COM files. Unfortunately I don't yet fully understand this format, so the instructions will be simple (pretty much the same as the .COM program) and more advanced concepts like relocations won't be covered. Being able to write a short MZ executable is needed for the PE executable which will be covered next, as they start off with an MZ at the beginning.

The header is as follows:

```
WORD magic;
WORD lastSize;
WORD nBlocks;
WORD nReloc;
WORD hdrSize;
WORD minAlloc;
WORD maxAlloc;
WORD ss;
WORD sp;
WORD checksum;
WORD ip;
WORD cs;
WORD relocPos;
WORD nOverlay;
```

Credit to http://www.delorie.com/djgpp/doc/exe/ for the info on the fields.

First and foremost, `magic` is a magic number that always has a value of `0x5A4D`, or as it appears in the file, `4D 5A`, representing the ASCII letters “MZ”.

`lastSize` is the number of bytes in the last block of the file. A 'block' is shorthand for 512 (`0x200`) bytes. Obviously if the file is less than a block long, this will just be the size of the file in bytes. A value of `0` means the entire last block is used.

`nBlocks` is the number of (not necessarily full) blocks in the file, with a minimum value of `1`. If `lastSize` isn't `0`, only `lastSize` bytes are used in the last block.

`nReloc` is the number of relocation entries in the file, if any.

`hdrSize` is the size of the header, in paragraphs. A 'paragraph' is 16 (`0x10`) bytes.

`minAlloc` and `maxAlloc`, respectively, are the minimum and maximum number of paragraphs of memory to reserve for the program.

`ss` and `sp` are the initial values of the Stack Segment (relative to the segment the program was loaded at) and Stack Pointer registers respectively.

`checksum` is the checksum of the file, unsurprisingly. It should be `0` if the file hasn't been corrupted or mangled in some way during transit.

`ip` and `cs` are the initial values of the Instruction Pointer and Code Segment (relative to the segment the program was loaded at) registers respectively. Use these to select where execution begins.

`relocPos` is the file offset of the relocation table, if one exists.

`nOverlay` is the overlay number of the file. I'm not exactly sure what an overlay number is... but a value of `0` seems to work.

To prepare for the PE format next, this MZ program will display a message saying “This program cannot be run in DOS mode.” and exit. This is because PE programs are made for 32-bit Windows and will not work on DOS, but they still have the same .exe extension. So PE programs all have a MZ 'stub' program at the beginning that is runnable by DOS, which typically states that the program needs Windows to run, can't run on DOS, or something to that effect.

For this we can use the COM program from before, with the ASCII string at the end modified to represent a different message. However, there is another thing we must do: in the COM program the `cs` and `ds` registers had the same value, which is why we could just supply `dx` with the offset of the string to use with `int 21h` (`int 21h`, service `09` uses `ds:dx` for the string). But `cs` and `ds` **aren't necessarily the same** in the MZ executable. So we must set `ds` manually. Since there are no `[reg]` codes for `cs` and `ds`, we can't do a simple `mov ds,cs`. Instead we push `cs` to the stack and pop it to `ds` at the beginning. Because of these two extra instructions, and the fact that the program **might not be loaded at `0x100`** any more, we also need to recalculate the offset of `msg`. So our assembly, and resulting machine code, looks like this:

```
ASSEMBLY                                MACHINE
========                                =======
push cs                                 0E
pop ds                                  1F
mov ah,09                               B4 09
mov dx,OFFSET msg                       BA 0F 00
int 21h                                 CD 21
mov ah,4Ch                              B4 4C
mov al,00                               B0 00
int 21h                                 CD 21
msg db “This program”\                  54 68 69 73 20 70 72 6F 67 72 61
“ cannot be run in DOS mode.$”          6D 20 63 61 6E 6E 6F 74 20 62 65
                                        20 72 75 6E 20 69 6E 20 44 4F 53
                                        20 6D 6F 64 65 2E 24
```

So, we have our machine code and now we need our header.

`magic` is going to be easy, just `4D 5A`.

`lastSize` requires the entire file to be completed first, so we mark it with `XX XX`.

`nBlocks` will be `1`, since there's no way such a small file can be over 512 bytes.

`nReloc` is `0` since we're not using relocations here.

`hdrSize` has yet to be determined; for now, mark it `XX XX`

`minAlloc` and `maxAlloc` can both be `0` as our program doesn't need any extra memory.

`ss` can be `0` and `sp` can be `B8` to give the stack some room.

`checksum` is obviously going to be `0`...

`ip` and `cs` are `0` because our program starts at the first instruction.

`relocPos` is `0` since we don't have a relocation table.

`nOverlay` is also `0`.

This gives us a header of:

```
4D 5A XX XX 01 00 00 00 – XX XX 00 00 00 00 00 00
B8 00 00 00 00 00 00 00 – 00 00 00 00
```

We can now fill in the `hdrSize` field. However it's not in bytes; it's in paragraphs, but the current size of the header is not a multiple of 16! Therefore we need to 'pad' the last four bytes with some value (I chose `00`):

```
4D 5A XX XX 01 00 00 00 – 02 00 00 00 00 00 00 00
B8 00 00 00 00 00 00 00 – 00 00 00 00 00 00 00 00
```

The new bytes have been shown in bold, in addition to the filled-in `hdrSize` field.

The header's almost finished. All we need to do now is to add in the machine code:

```
000 :4D 5A XX XX 01 00 00 00 – 02 00 00 00 00 00 00 00
010 :B8 00 00 00 00 00 00 00 – 00 00 00 00 00 00 00 00
020 :0E 1F B4 09 BA 0F 00 CD – 21 B4 4C B0 00 CD 21 54
030 :68 69 73 20 70 72 6F 67 – 72 61 6D 20 63 61 6E 6E
040 :6F 74 20 62 65 20 72 75 – 6E 20 69 6E 20 44 4F 53
050 :20 6D 6F 64 65 2E 24
```

Now we can fill in the `lastSize` member – using the file offsets on the left we can see the last byte in the file is at offset `0x56`, meaning the total number of bytes in the file is `0x57` (since the first byte is at offset `0`). The finished MZ executable hex is, with `lastSize` in bold:

```
000 :4D 5A 57 00 01 00 00 00 – 02 00 00 00 00 00 00 00
010 :B8 00 00 00 00 00 00 00 – 00 00 00 00 00 00 00 00
020 :0E 1F B4 09 BA 0F 00 CD – 21 B4 4C B0 00 CD 21 54
030 :68 69 73 20 70 72 6F 67 – 72 61 6D 20 63 61 6E 6E
040 :6F 74 20 62 65 20 72 75 – 6E 20 69 6E 20 44 4F 53
050 :20 6D 6F 64 65 2E 24
```

If you run the resulting program, you should see the message “This program cannot be run in DOS mode.”.

So far we have been working in 16-bit code and memory with nasty things like segmentation and ancient DOS conventions. The PE format is 32-bit (save for the MZ stub at the beginning) and thus uses 32-bit instructions. Apart from just having double-length operands, they have a few other differences as will be seen...

Chapter 3: The Portable Executable

As with the .COM chapter, we will begin with the assembly code for the program we wish to write. This will be in MASM form as it is a 32-bit, Windows assembler:

```
.386
.model flat,stdcall
option casemap:none
include <\masm32\include\windows.inc>
include <\masm32\include\kernel32.inc>
include <\masm32\include\user32.inc>
includelib <\masm32\lib\kernel32.lib>
includelib <\masm32\lib\user32.lib>

.data
msg        db    "Hello, World!",0
msgTitle   db    "Message",0

.code
start:
     push MB_ICONINFORMATION    ; 0x40
     push OFFSET msgTitle
     push OFFSET msg
     push NULL                  ;0
     call MessageBoxA
     push 0
     call ExitProcess
end start
```

32-bit machine code

The main difference between 16-bit and 32-bit x86 machine code is that all 16-bit operands are now 32-bit. So `r/m16` changes to `r/m32`, `imm16` changes to `imm32`, and so on (this means that the `[W]` bit denotes 8-bit or 32-bit values now). This isn't really hard at all to get used to, but one thing to remember is that **these instructions are, numerically, no different to their 16-bit counterparts**. That's right! If you look at the MASM opcode reference you'll see that the 16-bit and 32-bit opcodes are **exactly the same** (pretty much – there may be one or two exceptions for instructions not needed in 32-bit protected mode). So how do we use 16-bit instructions in 32-bit mode? The short answer is...we don't. In 32-bit protected mode the processor expects 32-bit instructions. The long answer is that you can use the **operand-size prefix** (`0x66`) and/or the **address-size prefix** (`0x67`) bytes **before** the opcode byte to tell the processor to treat the operands and/or address operand, respectively, as their 16-bit counterparts. So if you wanted to use `mov ax,[1234h]`, it would be `66 67 8B 06 34 12` as both the register (operand) and address displacement (address) are 16-bit. As another example, `mov dx,0FFFFh` would be `66 BA FF FF`, as it does not have a memory operand.

The second big difference is that the `[mod]`, `[reg]` and `[r/m]` fields of the mod r/m byte have different behaviour. Well, `[reg]` isn't that different, as you can see:

```
[reg]      [W]=0      [W]=1
000          al        eax
001          cl        ecx
010          dl        edx
011          bl        ebx       32-BIT MODE!
100          ah        esp
101          ch        ebp
110          dh        esi
111          bh        edi
```

However, the [mod] and [r/m] combinations are quite different...

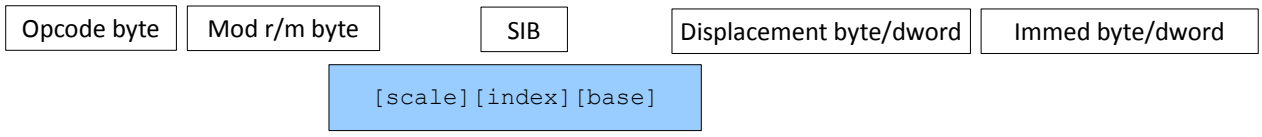
[mod]
 00 – Treat [r/m] as memory address
 01 – Treat [r/m] + disp8 as memory address
 10 – Treat [r/m] + disp32 as memory address
 11 – Treat [r/m] as another [reg] field

[r/m]
 000 - eax
 001 - ecx
 010 - edx
 011 - ebx
 100 - SIB **unless** [mod] is 11, in which case esp
 101 - ebp **unless** [mod] is 00, in which case disp32
 110 - esi
 111 - edi

The following table shows the new combinations for the value of the operand:

[r/m] -> [mod] V	000	001	010	011	100	101	110	111
00	[eax]	[ecx]	[edx]	[ebx]	[SIB]	[disp32]	[esi]	[edi]
01	[eax+ disp8]	[ecx+ disp8]	[edx+ disp8]	[ebx+ disp8]	[SIB+ disp8]	[ebp+ disp8]	[esi+ disp8]	[edi+ disp8]
10	[eax+ disp32]	[ecx+ disp32]	[edx+ disp32]	[ebx+ disp32]	[SIB+ disp32]	[ebp+ disp32]	[esi+ disp32]	[edi+ disp32]
11	eax	ecx	edx	ebx	esp	ebp	esi	edi

Now what on earth is a SIB? It stands for Scale Index Byte, and is another byte that can come after the mod r/m byte. So the new 32-bit instruction format is:



The SIB lets you do things like mov eax, [esi+ecx*4], aka operands of the form:

```
base+(index*scale)
```

While it's not used in our current program, it is still worth knowing about. The [scale] field is **2 bits wide** and denotes a scale of:

[scale]
  00 -   1
  01 -   2
  10 -   4
  11 -   8

[index] is **3 bits wide** and can be the following:

```
[index]
 000 - eax
 001 - ecx
 010 - edx
 011 - ebx
 100 - 0
 101 - ebp
 110 - esi
 111 - edi
```

Lastly, [base] is also **3 bits wide** and behaves almost exactly like the [r/m] field (note it depends on [mod]):

```
[base]
 000 - eax
 001 - ecx
 010 - edx
 011 - ebx
 100 - esp
 101 - ebp unless [mod] is 00, in which case disp32
 110 - esi
 111 - edi
```

Unfortunately if I put a table here summarising all this it would be huge, but the information above is all that is needed.

Examples

Time to get familiar with 32-bit instructions...

1) mov ebp,esp

Starting off easy here. Opcode to use can be either 89 or 8B, depending on the order of the operands; I'll use 89 for this. The [D] bit is 0, denoting [reg] as the source, so our [reg] = 100 (esp). To get ebp as the destination we'll need a [mod] of 11 and a [r/m] of 101. So our mod r/m byte is 11100101, or E5 in hex, giving a result of **89 E5**.

2) mov dword ptr [ebp+8],100h

The opcode to use for this is C7. It has the /0 thing so we've got our [reg] field there already. For [r/m] we could treat 8 as a disp32, but we might as well not since it's small enough to fit in a disp8. If we look in the table for [ebp+disp8] we find a [mod] of 01 and a [r/m] of 101, giving a mod r/m byte of 01000101 = 45 hex. All we need to do now is append the disp8 (8) and imm32 (0x100) giving a final result of **C7 45 08 00 10 00 00**.

3) add eax,[esi+ecx*4]

SIB time! Don't worry, it's not that bad. First of all, the opcode we want is 03. The [D] bit's 1, because [reg] is the destination. So [reg] is going to be 000 for eax. The second operand is of the form [SIB], and a quick look at the table tells us that'll need [mod] = 00 and [r/m] = 100. So our mod r/m byte is 00000100 = 04 hex. For our SIB, esi is the base, ecx is the index and 4 is the scale. Thus [scale] = 10, [index] = 001 and [base] = 110, giving a final SIB of 10001110 = 8E hex. Put it all together and you get **03 04 8E**.

4) mov edx,[ebp+eax*2]

The opcode for this is 8B. [reg], being the destination, is 010 for edx. The second operand is [SIB]

again, so go to the table and use a [mod] of 00 and a [r/m] of 100. Then, for the SIB – wait a minute, we can't have a base of ebp because our [mod] is 00! Is it impossible then? Not at all, you just have to work around it. Instead of having the second operand as [SIB], have it as [SIB + disp8], which avoids a [mod] of 00, letting you use ebp as a base. The disp8 can just be 0, which may waste some CPU time for the addition but it is the only way to accomplish this, unfortunately. So our [mod] is 01, [reg] is 010, [r/m] is 100, [scale] is 01, [index] is 000, and our [base] is 101. Phew!

Putting it all together, including the disp8, gives **8B 54 45 00**.

We're not ready to write the executable yet though, as first we must understand the actual format and the huge mass of headers at the beginning. Oh boy...

Structure of the Portable Executable

The basic structure is as follows:

```
MZ 'DOS' header + stub program
PE signature
COFF header
'Optional' header
Section table
Sections containing code/data/whatever
```

Firstly...

The DOS header and stub program

You thought you'd seen the last of DOS? Not so! As mentioned in the MZ exe chapter, the PE has a DOS MZ program at the start for compatibility reasons. Its header is very similar to the original DOS header used in the last chapter, but with **these extra fields**:

```
WORD reserved1[4];
WORD oem_id;
WORD oem_info;
WORD reserved2[10];
DWORD e_lfanew;
```

Just set them all to 0, **except** for e_lfanew which is very important because it holds the **file offset** of the PE signature.

There is another requirement: the relocPos field has to be 0x40 in a PE file. Let's grab the MZ executable from the previous chapter and modify it for the PE – we will have to modify the lastSize and hdrSize fields accordingly:

```
0000 :4D 5A 77 00 01 00 00 00 - 04 00 00 00 00 00 00 00
0010 :B8 00 00 00 00 00 00 00 - 40 00 00 00 00 00 00 00
0020 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0030 :00 00 00 00 00 00 00 00 - 00 00 00 00 XX XX XX XX
0040 :0E 1F B4 09 BA 0F 00 CD - 21 B4 4C B0 00 CD 21 54
0050 :68 69 73 20 70 72 6F 67 - 72 61 6D 20 63 61 6E 6E
0060 :6F 74 20 62 65 20 72 75 - 6E 20 69 6E 20 44 4F 53
0070 :20 6D 6F 64 65 2E 24
```

The modified and newly added data is shown in bold. Note that e_lfanew has been marked as unknown with X's for the moment. If the program is run under DOS then it'll skip the fields it doesn't know about because of the hdrSize member; but if run on Windows the PE loader will recognise the special value of relocPos and will proceed to load it as a PE.

The PE Signature

Coming right after the DOS stub is the PE signature, which is used by Windows to identify the file as an actual PE. It is a **DWORD** so it is 4 bytes wide, and has a value of `0x004550`. This may seem fairly arbitrary but it appears in the file as `50 45 00 00`, corresponding to the ASCII letters "PE" followed by two NULLs.

If I am correct then the PE signature must be aligned on a 16-byte boundary. I am not certain of this but it is a good idea to do so anyway, so we will :)

If we pad the rest of the DOS stub with zeroes we can put the PE signature after like so:

```
0000 :4D 5A 77 00 01 00 00 00 – 04 00 00 00 00 00 00 00
0010 :B8 00 00 00 00 00 00 00 – 40 00 00 00 00 00 00 00
0020 :00 00 00 00 00 00 00 00 – 00 00 00 00 00 00 00 00
0030 :00 00 00 00 00 00 00 00 – 00 00 00 00 80 00 00 00
0040 :0E 1F B4 09 BA 0F 00 CD - 21 B4 4C B0 00 CD 21 54
0050 :68 69 73 20 70 72 6F 67 - 72 61 6D 20 63 61 6E 6E
0060 :6F 74 20 62 65 20 72 75 - 6E 20 69 6E 20 44 4F 53
0070 :20 6D 6F 64 65 2E 24 00 – 00 00 00 00 00 00 00 00
0080 :50 45 00 00
```

Notice how we can now set the value of `e_lfanew`, which contains the file offset of the PE signature (in this case `0x80`).

The COFF header

Right after the PE signature is the COFF header. If you're wondering, COFF stands for Common Object File Format and is used in other files like .OBJ's produced by compilers. Anyway, the COFF header has the following fields:

```
WORD machine;
WORD numberOfSections;
DWORD timeDateStamp;
DWORD pointerToSymbolTable;
DWORD numberOfSymbols;
WORD sizeOfOptionalHeader;
WORD characteristics;
```

`machine` is a number that represents the processor architecture (machine) the executable is meant for. A full list of values is available in the Windows headers (`WinNT.h`) or on the Internet, but since this entire text is for x86-compatible processors, we will use the value `0x14C`, which represents an intel 386-compatible processor.

`numberOfSections` is, unsurprisingly, the number of sections in the file. Sections will be covered later on, but they are basically areas of code or data that are mapped into memory one by one by the PE loader. We will have **3** sections here to keep things relatively simple, but some executables have many more.

`timeDateStamp` is the time and date the executable was created, in UTC format. This isn't essential and unless you really want to count up the number of seconds since January 1, 1970 to the present day (or just use a converter, but still...) we will leave this field as `0`.

Once upon a time, `pointerToSymbolTable` and `numberOfSymbols` represented the file offset of the symbol table and the number of symbols, respectively. However the PE specification states that this information is now deprecated and they should be left as `0`.

`sizeOfOptionalHeader` is the size, in bytes, of the Optional Header. The Optional Header comes next

after the COFF header, and its full size is `0xE0` bytes.

`characteristics` describes properties of the executable as a whole. This can be things like if it has or lacks relocations, is a DLL, and so forth. DLL's are actually exe's for pretty much all intents and purposes, and they follow the PE format. However they differ in several areas, including this field. For our executable we shall have the following flags:
`IMAGE_FILE_RELOCS_STRIPPED (0x01)`, since we don't use relocations here
`IMAGE_FILE_EXECUTABLE_IMAGE (0x02)`, as this is required for the executable to actually run
`IMAGE_FILE_32BIT_MACHINE (0x100)`, as the executable is made for a 32-bit machine and uses 32-bit instructions

Binary OR'ing these flags together gives a final `characteristics` of `0x103`. By the way, when we speak of an *executable image*, we refer to the executable **as it appears loaded into memory**.

So far, the nice, modern, non-DOS part of the file is:

```
0080 :50 45 00 00 4C 01 03 00 - 00 00 00 00 00 00 00 00
0090 :00 00 00 00 E0 00 03 01
```

The newly added COFF header is shown in bold.

Which brings us to the Optional Header! Prepare yourself, as I wasn't joking when I said its full size is `0xE0` (124) bytes...

The Optional Header

The Optional Header's name is a little misleading as it's actually required as part of PE files. What gives it its name is that it *is* optional in .OBJ files; just not in EXE's or DLL's.

Before we begin it is important to know several things. Firstly, when an executable (referring to both EXEs and DLLs) is loaded, its sections are mapped into memory **starting at a specific address**. This address is known as the **image base**. The image base is actually a **preferred** address; if the loader is unable to do this, perhaps because the address is already occupied by something else, it can use a **different** address that is available (however this only tends to happen with DLLs since EXEs are usually there first). For this reason using **absolute** memory addresses to refer to things isn't guaranteed to work.

Instead a form of **relative address** is used by fields in the PE headers. This is known as an **RVA** – Relative Virtual Address. An RVA is simply an offset from the image base; that is, it's a memory address, but measured relative to the image base. As an example, suppose an EXE has an image base of `0x400000` (this is actually the default value). One of its sections says it wants to be loaded at RVA `0x1000`. Since an RVA is just an offset from the image base, the loader will put the section at address `0x400000 + 0x1000 = 0x401000` in memory.

Obviously this would also work if we replace the EXE with a DLL which can't be loaded at its preferred image base – say the loader managed to place it at `0x800000`; the section would still be loaded at the correct address of `0x801000`. This would not be the case if RVAs were not used, as the section would be trying to load itself into an already-occupied memory address.

So, ready for the Optional Header fields? Here they are:

```
WORD magic;
BYTE majorLinkerVersion;
BYTE minorLinkerVersion;
DWORD sizeOfCode;
DWORD sizeOfInitData;
```

```
DWORD sizeOfUninitData;
DWORD addrOfEntryPoint;
DWORD baseOfCode;
DWORD baseOfData;
DWORD imageBase;
DWORD sectionAlignment;
DWORD fileAlignment;
WORD majorOSversion;
WORD minorOSversion;
WORD majorImgVersion;
WORD minorImgVersion;
WORD majorSubsystemVersion;
WORD minorSubsystemversion;
DWORD reserved;
DWORD sizeOfImage;
DWORD sizeOfHeaders;
DWORD checksum;
WORD subsystem;
WORD dllCharacteristics;
DWORD sizeOfStackReserve;
DWORD sizeOfStackCommit;
DWORD sizeOfHeapReserve;
DWORD sizeOfHeapCommit;
DWORD loaderFlags;
DWORD numOfRvaAndSizes;
IMAGE_DATA_DIRECTORY dataDirectory[numOfRvaAndSizes];
```

`magic` is a magic number that is always `0x10B` for 32-bit systems.

`majorLinkerVersion` and `minorLinkerVersion` are fairly self-explanatory. Combined, they represent the version of the linker used when linking the program. Since we're doing this the 'fun' way ('fun' being open to interpretation ;P) and not doing any compiling or linking, we can set this to `0`.

`sizeOfCode` is the total size, in bytes, of the section(s) containing executable code (machine instructions). `sizeOfInitData` is the same but for those containing initialised data, and `sizeOfUninitData` for uninitialised data. Since we don't yet know the size of these sections (or even what exactly a section is) we will mark these with X's to show they are currently unknown.

`addrOfEntryPoint` is the RVA of the entry point in our program. The entry point is the address where execution should begin (ie whatever memory location the first instruction in our program corresponds to). We don't yet know this either, so it will be marked as such.

`baseOfCode` is the RVA of where our code section will be in memory. Since we don't know much about sections yet, this will be left unknown for now.

`baseOfData` is the same but for our first data section. This will also be left unknown and filled in later.

`imageBase` is the image base of the executable; aka its preferred load address. For our EXE we will use the default value of `0x400000`. According to the specification, it has to be a multiple of 64k (`0x10000`).

`sectionAlignment` is the alignment of sections in memory. This means that the load addresses of the sections must be multiples of this number. On x86 compatible processors this can be no less than the page size (4 Kb), and to save space we will use this as it is the lowest value. So our sections' addresses can only be multiples of 4 kb, or `0x1000`.

`fileAlignment` is the alignment of sections in the file. This must be a power of 2 between 512 (`0x200`) and 64k (`0x10000`). Since we want as small a file as is possible, we will choose 512 (`0x200`).

Of the next 6 fields, only 2 really matter: `majorOSversion`, which is the major version number of the required OS, and `majorSubsystemVersion` which is the same but for the subsystem (eg Windows GUI, Windows CUI, native etc). Values of 4 seem to work with these fields. The `ImgVersion`'s are the versions of the executable, so they are defined by the creator. They, and the rest of the `minor` fields, can be `0`.

`reserved` must be `0`.

`sizeOfImage` is the **virtual** size of the executable image in memory. This means that it is the combined size of the headers and sections, **as a multiple of `sectionAlignment`**. We don't yet know enough information to complete this field, so we mark it as unknown with X's.

`sizeOfHeaders` is the size of all of the headers in the file (including the section table), rounded up to **a multiple of `fileAlignment`**. The sections themselves start at this offset in the file. However we don't yet know this currently, so we'll mark this with X's too.

`checksum`, like all good checksums, should be `0`.

`subsystem` represents the subsystem the program uses; important values are `IMAGE_SUBSYSTEM_NATIVE (0x01)`, used for things like device drivers, `IMAGE_SUBSYSTEM_WINDOWS_GUI (0x02)`, used for graphical Windows applications, and `IMAGE_SUBSYSTEM_WINDOWS_CUI (0x03)`, used for console-mode Windows applications. Since our program displays a message box, it's going to need `IMAGE_SUBSYSTEM_WINDOWS_GUI`, and so will have a value of `0x02`.

`dllCharacteristics` is only used for DLLS, so we can use a value of `0` for this field.

`sizeOfStackReserve` and `sizeOfStackCommit` are how much memory to reserve for the stack and how much to actually commit, respectively. Basically the memory is reserved for the stack but only a certain amount is made available. If this memory runs out then the reserve can be made available one page (4k) at a time until it runs out too. We will have a reserve of `0x10000` (16 pages) and a commit of `0x1000` (1 page) for our program.

`sizeOfHeapReserve` and `sizeOfHeapCommit` follow the same principle, except that it is for the size of the heap instead. For simplicity we will use the same respective values here as for the Stack fields.

`loaderFlags` is a reserved field and needs to be `0`.

`numOfRvaAndSizes` is the number of entries in the Data Directory, and we will use its full value which is 16 (`0x10`).

Finally, `dataDirectory` is an array of `IMAGE_DATA_DIRECTORY`'s which provides information on special data structures used by Windows. An `IMAGE_DATA_DIRECTORY` is an 8-byte structure composed of:

```
DWORD virtualAddress;
DWORD size;
```

`virtualAddress` is the RVA of the data structure in memory. `size` is the size, in bytes, of the relevant data.

Each entry of the Data Directory has a specific data structure associated with it. Of use to us are members `[1]` and `[12]`, which are `IMAGE_DIRECTORY_ENTRY_IMPORT` and `IMAGE_DIRECTORY_ENTRY_IAT`. The first one holds information about **imported functions** from **DLLs**, in our case `MessageBoxA` from `user32.dll` and `ExitProcess` from `kernel32.dll`. The second is the **Import Address table**, or IAT. This is where the addresses of said functions are placed for our program to use. We will go into much more detail on these later (heck, we have to *write* them) but for now we will mark both entries completely unknown with X's. The rest of the entries are `0`.

In summary, here is our PE file from offset `0x80` so far, with the fields of the Optional Header shown in bold, and the Data Directory part in blue:

```
0080 :50 45 00 00 4C 01 03 00 - 00 00 00 00 00 00 00 00
0090 :00 00 00 00 E0 00 03 01 - 0B 01 00 00 XX XX XX XX
00A0 :XX XX XX XX XX XX XX XX - XX XX XX XX XX XX XX XX
00B0 :XX XX XX XX 00 00 40 00 - 00 10 00 00 00 02 00 00
00C0 :04 00 00 00 00 00 00 00 - 04 00 00 00 00 00 00 00
00D0 :XX XX XX XX XX XX XX XX - 00 00 00 00 02 00 00 00
00E0 :00 00 01 00 00 10 00 00 - 00 00 01 00 00 10 00 00
00F0 :00 00 00 00 10 00 00 00 - 00 00 00 00 00 00 00 00
0100 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0110 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0120 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0130 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0140 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0150 :00 00 00 00 00 00 00 00 - XX XX XX XX XX XX XX XX
0160 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0170 :00 00 00 00 00 00 00 00
```

Which *finally* brings us to the Section Table.

The Section Table

As mentioned, a **section** is just a **chunk of code or data** in the executable. They're organised into sections to avoid having code and data all over the place and also because **individual sections can have different characteristics**. So, for example, a code section could be marked as executable (obviously) and readable, but perhaps not writeable for more security. Or perhaps, a data section could be marked as non-executable for the same reason. Sections, as mentioned, have to be **aligned** in both memory and in the file. In memory they are aligned to the value of `sectionAlignment` in the Optional Header, and in the file / on disk they are aligned to `fileAlignment`. The number of sections in the file is given near the start of the COFF header, as the `numberOfSections` field we set to 3.

The **section table** is the final part of the header structure and holds information about the different sections in the file. It is composed of **section headers**, with one header per section. The format of a section header is as follows:

```
BYTE name[8];
DWORD virtualSize;
DWORD virtualAddress;
DWORD sizeOfRawData;
DWORD pointerToRawData;
DWORD pointerToRelocations;
DWORD pointerToLineNumbers;
WORD numberOfRelocations;
WORD numberOfLineNumbers;
DWORD characteristics;
```

`name` is an ASCII string with a maximum length of 8 characters. It usually begins with a full stop and is usually one of these:

`.text` – used for the executable code section
`.data` – used for initialised, writeable data
`.rdata` – used for read-only data, such as string literals
`.idata` – Import Data; used for the Import and Import Address Table.
`.edata` – Export Data; used for exports (mainly used by DLLs)
`.bss` – Uninitialised data

In reality, however, you can actually call them whatever you want. They are really only meant for humans to read. If the name is less than 8 characters long, the rest of this field is padded with zeroes.

`virtualSize` is the size of the actual data. This is not a rounded or aligned value, unlike `sizeOfRawData`. Uninitialised data sections do not need section data, but in this case this field will be the amount of data in the section *once loaded*.

`virtualAddress` is the RVA of the section in memory. It must be a multiple of `sectionAlignment`, in this case a multiple of `0x1000`.

`sizeOfRawData` is the size of the section data; the only problem is, it **must be a multiple of `fileAlignment`**, in our case a multiple of `0x200`.

`pointerToRawData` is the **file offset** of the section data in the file. This **must also be a multiple of `fileAlignment`**, giving an important restriction on the sections in our file: they must all start at multiples of 512 bytes. Unfortunately with such a small assembly program there will be a lot of empty space in the file, but this is one of the things you have to live with :S

`pointerToRelocations` is the file offset of relocation data for the section. Relocations are not needed in our EXE because they are pretty much always loaded at their preferred image base, so for all sections this field will be `0`, denoting no relocations.

`pointerToLineNumbers` is very much like `pointerToSymbolTable` in the COFF header; it's deprecated and so should be `0`.

`numberOfRelocations` and `numberOfLineNumbers` are self explanatory and will both be `0` for the section headers in our file.

Now, `characteristics` is the really important field here. It is a combination of flags denoting specific properties the section should have once loaded / mapped into memory. Here are some common values:

`IMAGE_SCN_CNT_CODE (0x20)` – Section contains machine instructions
`IMAGE_SCN_CNT_INITIALISED_DATA (0x40)` – Section contains initialised data
`IMAGE_SCN_CNT_UNINITIALISED_DATA (0x80)` – Section contains uninitialised data
`IMAGE_SCN_MEM_EXECUTE (0x20000000)` – Section is executable
`IMAGE_SCN_MEM_READ (0x40000000)` – Section is readable (I cannot think of why one would omit this characteristic...)
`IMAGE_SCN_MEM_WRITE (0x80000000)` – Section is writeable (aka not read-only)

According to the specification, the sections and their headers must come in ascending order of `virtualAddress`.

Our section table

First of all, let's decide what sections we're going to need. Obviously we're going to need a code section for our machine instructions (`.text`). We'll also need a read-only data section (`.rdata`) to hold our string literals (although for this program it's not really that important whether or not it's read-only. Heck, we could even store the strings in the code section if we wanted, so long as we make sure they're not accidentally executed). Finally, we'll need a section for our import info and IAT (`.idata`). Since the only other thing we currently know about the sections are their characteristics, we will have to leave several other fields unknown until we have the sections themselves.

First of all, the `.text` section will have characteristics `IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ = 0x60000020`.

Next, the `.rdata` section will have characteristics `IMAGE_SCN_CNT_INITIALISED_DATA | IMAGE_SCN_MEM_READ = 0x40000040`.

And the `.idata` section will have characteristics `IMAGE_SCN_CNT_INITIALISED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE = 0xC0000040`.

So our file from offset `0x170` with the section table in bold looks like this:

```
0170 :00 00 00 00 00 00 00 00 – 2E 74 65 78 74 00 00 00
0180 :XX XX XX XX XX XX XX XX – XX XX XX XX XX XX XX XX
0190 :00 00 00 00 00 00 00 00 – 00 00 00 00 20 00 00 60
01A0 :2E 72 64 61 74 61 00 00 – XX XX XX XX XX XX XX XX
01B0 :XX XX XX XX XX XX XX XX – 00 00 00 00 00 00 00 00
01C0 :00 00 00 00 40 00 00 40 – 2E 69 64 61 74 61 00 00
01D0 :XX XX XX XX XX XX XX XX – XX XX XX XX XX XX XX XX
01E0 :00 00 00 00 00 00 00 00 – 00 00 00 00 40 00 00 C0
```

The actual sections come right after the section table. However, remember that they must be aligned according to `fileAlignment`. Luckily the next multiple of `0x200` is just round the corner (16 bytes away) so all we have to do is pad it with zeroes:

```
01B0 :XX XX XX XX XX XX XX XX – 00 00 00 00 00 00 00 00
01C0 :00 00 00 00 40 00 00 40 – 2E 69 64 61 74 61 00 00
01D0 :XX XX XX XX XX XX XX XX – XX XX XX XX XX XX XX XX
01E0 :00 00 00 00 00 00 00 00 – 00 00 00 00 40 00 00 C0
01F0 :00 00 00 00 00 00 00 00 – 00 00 00 00 00 00 00 00
0200 :
```

Great! But before we start working on the first section, we should realise that the headers are now finished! This means we can go back to the Optional Header and finally fill in the `sizeOfHeaders` field. The `sizeOfHeaders` field is now `0x200`:

```
00D0 :XX XX XX XX 00 02 00 00 – 00 00 00 00 02 00 00 00
```

We can now work on the first section.

The `.text` section

For this we will need to translate the assembly instructions from our program. They are repeated here:

```
push MB_ICONINFORMATION    ; 0x40
push OFFSET msgTitle
push OFFSET msg
push NULL                  ;0
call MessageBoxA
push 0
call ExitProcess
```

If you did the conversion correctly you should have ended up with (the equivalent of) the following machine code:

```
6A 40 68 XX XX XX XX 68 - XX XX XX XX 6A 00 FF 15
XX XX XX XX 6A 00 FF 15 - XX XX XX XX
```

Obviously we don't yet know the `OFFSET`'s or function addresses, so we leave them unknown for now.

And that's pretty much it for our code section...it's a small program, so it's a small code section. If we count up the bytes we get a total of `0x1C`; this value goes in the relevant section header's `virtualSize` field.

Since the next section will also have to be file-aligned, the next available offset would be `0x400` which means we will have to pad the **entire** rest of the section on-disk with zeroes. This is also true with the other sections, since their size also needs to be a multiple of `fileAlignment`.

```
0200 :6A 40 68 XX XX XX XX 68 - XX XX XX XX 6A 00 FF 15
0210 :XX XX XX XX 6A 00 FF 15 - XX XX XX XX 00 00 00 00
0230 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00

                <lots and lots of 00's>

03E0 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
03F0 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0400 :
```

We can now also update the `.text` section header's `sizeOfRawData` and `pointerToRawData` fields with `0x200` for both:

```
0170 :00 00 00 00 00 00 00 00 - 2E 74 65 78 74 00 00 00
0180 :1C 00 00 00 XX XX XX XX - 00 02 00 00 00 02 00 00
0190 :00 00 00 00 00 00 00 00 - 00 00 00 00 20 00 00 60
```

The `.rdata` section

This section is even smaller. All we have to do is put in the string literals, converted from ASCII to hex:

```
0400 :48 65 6C 6C 6F 2C 20 57 – 6F 72 6C 64 21 00 4D 65
0410 :73 73 61 67 65 00
```

The `virtualSize` of this section is `0x16` and the `pointerToRawData` is obviously `0x400`. The `sizeOfRawData` is also `0x200` once we pad up to offset `0x600`. We should update the `.rdata` section header with these values:

```
01A0 :2E 72 64 61 74 61 00 00 – 16 00 00 00 XX XX XX XX
01B0 :00 02 00 00 00 04 00 00 – 00 00 00 00 00 00 00 00
01C0 :00 00 00 00 40 00 00 40 – 2E 69 64 61 74 61 00 00
```

The `.idata` section

Remember the Import data directory and IAT from the Optional Header? In the file, they are located here. Here's how it works: for **each DLL** the program requires, there is an `IMAGE_IMPORT_DESCRIPTOR` structure describing it and the functions to import from it. They form an array of `IMAGE_IMPORT_DESCRIPTOR`'s, terminated by a descriptor with **all fields having a value of 0**. The `IMAGE_IMPORT_DESCRIPTOR` is as follows:

```
DWORD originalFirstThunk;
DWORD timeDateStamp;
DWORD forwarderChain;
DWORD name;
DWORD firstThunk;
```

I'll break the tradition of going through the fields in order, because the first and the last are of great importance.

`timeDateStamp` is another UTC-format time / date value that isn't really important and can be left at `0`.

`forwarderChain` is used in an advanced dynamic linking topic which is currently beyond the scope of this text, so it will be `0`.

`name` is an RVA to a null-terminated ASCII string which is the name of the DLL the descriptor is referring to (.dll extension included in the string).

Now, for **every function** (technically not necessarily a function but it will do for our purposes) that is exported by the DLL there is an `IMAGE_IMPORT_BY_NAME` structure, which is defined:

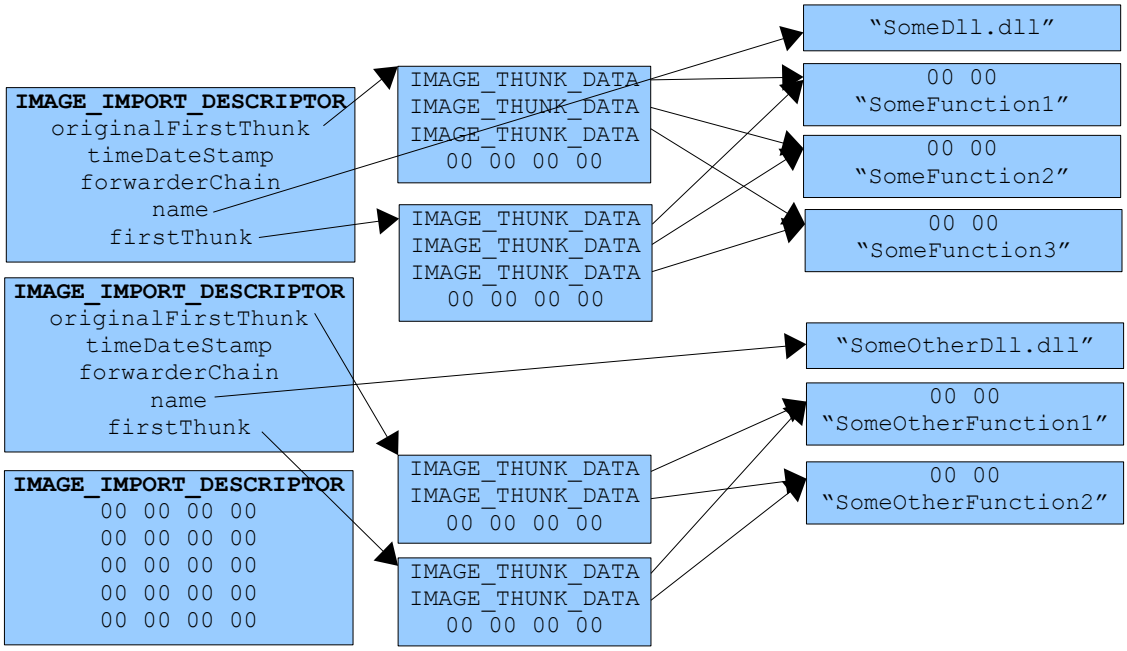
```
WORD hint;
BYTE name[1];
```

`hint` isn't needed and is an index into the DLL's export table. This will be `0` for instances of this structure in this program. The field of interest is `name` which isn't actually an array of length 1. `name` is the name of the function in question, and is a null-terminated ASCII string.

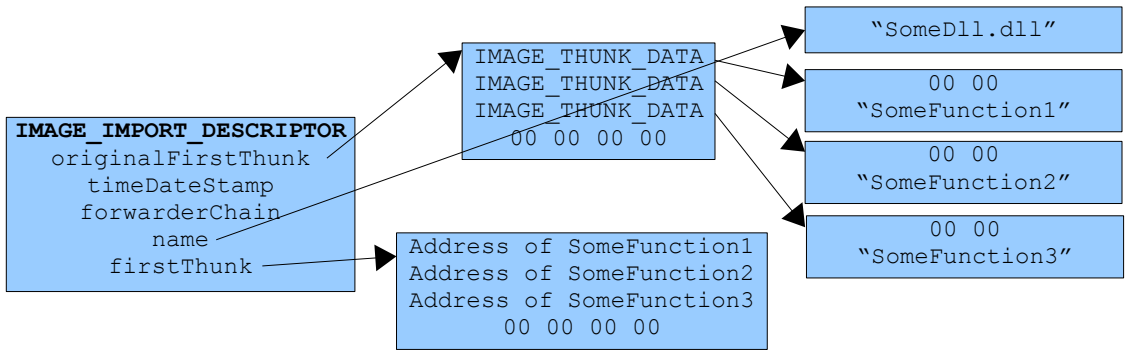
For every DLL there is an array of `DWORD`s (the actual name is `IMAGE_THUNK_DATA`), each holding the RVA of an `IMAGE_IMPORT_BY_NAME` for an imported function. The array is terminated by a null entry (a `DWORD` of value `0`). **The `originalFirstThunk` member of the `IMAGE_IMPORT_DESCRIPTOR` holds the RVA of this array.**

There is actually another array of `IMAGE_THUNK_DATA`'s that is *identical to the first*. The `firstThunk`

member holds the RVA of this array. Here's a diagram of the whole thing, to help visualise it:



So why on earth is there another array that does exactly the same thing as the first? Well, the combined `IMAGE_THUNK_DATA` arrays pointed to by the `firstThunk`s of all of the `IMAGE_IMPORT_DESCRIPTOR`'s **makes up the Import Address Table**! When the executable is loaded, Windows goes through the Import Table containing the import descriptors. It loads the DLLs one by one into the process address space by looking at the `name` fields, and for **each descriptor** looks at the function names pointed to by `firstThunk`'s `IMAGE_THUNK_DATA` array. It then **looks for the functions** in the relevant DLL and **replaces the `IMAGE_THUNK_DATA`'s with the addresses of those functions**, ready for the program's code to call. However it's always a good idea to have the RVAs to the function names still available, and the `originalFirstThunk`'s thunk data array achieves this purpose, just in case anything goes wrong. This is what the first DLL's import descriptor from the above diagram will look like once everything's ready:



Our Import Table

It may seem confusing at first, but all it takes is some getting used to. In our application we only have 2 DLLs, `kernel32` and `user32`. From `kernel32` we only import one function, `ExitProcess`. We also only import one function from `user32`, `MessageBoxA`. Let's write our `IMAGE_IMPORT_DESCRIPTOR` array first to begin our `.idata` section:

```
0600 :XX XX XX XX 00 00 00 00 - 00 00 00 00 XX XX XX XX
0610 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0620 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00
```

Unfortunately we can't add the details till we're done with the array. So now that we've written the `IMAGE_IMPORT_DESCRIPTOR` array, we need to reserve space for the `IMAGE_THUNK_DATA` arrays. How many members will we need? Well, there are two arrays per DLL, each with one `IMAGE_THUNK_DATA` per function from that DLL. Since we only import one function from each DLL, all four arrays will only contain one `IMAGE_THUNK_DATA` and one null entry:

```
0600 :XX XX XX XX 00 00 00 00 - 00 00 00 00 XX XX XX XX
0610 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0620 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00 XX XX XX XX
0640 :00 00 00 00 XX XX XX XX - 00 00 00 00 XX XX XX XX
0650 :00 00 00 00 XX XX XX XX - 00 00 00 00
```

Next we add the DLL names:

```
0600 :XX XX XX XX 00 00 00 00 - 00 00 00 00 XX XX XX XX
0610 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0620 :XX XX XX XX XX XX XX XX - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00 XX XX XX XX
0640 :00 00 00 00 XX XX XX XX - 00 00 00 00 XX XX XX XX
0650 :00 00 00 00 XX XX XX XX - 00 00 00 00 6B 65 72 6E
0660 :65 6C 33 32 2E 64 6C 6C - 00 75 73 65 72 33 32 2E
0670 :64 6C 6C 00
```

And finally, we add the `IMAGE_IMPORT_BY_NAME`'s:

```
0650 :00 00 00 00 XX XX XX XX - 00 00 00 00 6B 65 72 6E
0660 :65 6C 33 32 2E 64 6C 6C - 00 75 73 65 72 33 32 2E
0670 :64 6C 6C 00 00 00 45 78 - 69 74 50 72 6F 63 65 73
0680 :73 00 00 00 4D 65 73 73 - 61 67 65 42 6F 78 41 00
```

So we should now be ready to fill everything in. But wait! The things we need to fill in are RVAs - and we don't yet know the RVA of this section, or any others! Better determine them now...

First of all, the sections need RVAs compliant with `sectionAlignment`. We can afford to have the sections the minimum distance of 1 page (`0x1000` bytes) apart, as the sections themselves aren't nearly this big. So, `.text` can be at RVA `0x1000`, `.rdata` at `0x2000` and `.idata` at `0x3000`. Remember that the sections and their headers have to be in ascending order of RVA. We also know that `.idata`'s `sizeOfRawData` is `0x200` (yes, we need to pad it with zeroes) and its file offset is `0x600`. We can also see that its `virtualSize` will be `0x90`. Let's update the section headers to reflect all this:

```
0170 :00 00 00 00 00 00 00 00 - 2E 74 65 78 74 00 00 00
0180 :1C 00 00 00 00 10 00 00 - 00 02 00 00 00 02 00 00
0190 :00 00 00 00 00 00 00 00 - 00 00 00 00 20 00 00 60
01A0 :2E 72 64 61 74 61 00 00 - 16 00 00 00 00 20 00 00
01B0 :00 02 00 00 00 04 00 00 - 00 00 00 00 00 00 00 00
01C0 :00 00 00 00 40 00 00 40 - 2E 69 64 61 74 61 00 00
01D0 :90 00 00 00 00 30 00 00 - 00 02 00 00 00 06 00 00
01E0 :00 00 00 00 00 00 00 00 - 00 00 00 00 40 00 00 C0
```

The section table is now finished! Now we know the RVA of this section we can fill in the missing fields. Let's start with those in the Import Descriptors:

```
0600 :3C 30 00 00 00 00 00 00 - 00 00 00 00 5C 30 00 00
0610 :4C 30 00 00 44 30 00 00 - 00 00 00 00 00 00 00 00
0620 :69 30 00 00 54 30 00 00 - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00 XX XX XX XX
0640 :00 00 00 00 XX XX XX XX - 00 00 00 00 XX XX XX XX
0650 :00 00 00 00 XX XX XX XX - 00 00 00 00 6B 65 72 6E
0660 :65 6C 33 32 2E 64 6C 6C - 00 75 73 65 72 33 32 2E
0670 :64 6C 6C 00 00 00 45 78 - 69 74 50 72 6F 63 65 73
0680 :73 00 00 00 4D 65 73 73 - 61 67 65 42 6F 78 41 00
```

For clarity the Import Descriptor for `kernel32.dll` has been highlighted in yellow and that of `user32.dll` in green.

Since file offset `0x600` now corresponds to RVA `0x3000`, all one needs to do is find the offset from `0x600` and add it to `0x3000` to get the RVAs of the various fields. Next we give the `IMAGE_THUNK_DATA`'s the RVAs to the function names:

```
0600 :3C 30 00 00 00 00 00 00 - 00 00 00 00 5C 30 00 00
0610 :4C 30 00 00 44 30 00 00 - 00 00 00 00 00 00 00 00
0620 :69 30 00 00 54 30 00 00 - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00 74 30 00 00
0640 :00 00 00 00 82 30 00 00 - 00 00 00 00 74 30 00 00
0650 :00 00 00 00 82 30 00 00 - 00 00 00 00 6B 65 72 6E
0660 :65 6C 33 32 2E 64 6C 6C - 00 75 73 65 72 33 32 2E
0670 :64 6C 6C 00 00 00 45 78 - 69 74 50 72 6F 63 65 73
0680 :73 00 00 00 4D 65 73 73 - 61 67 65 42 6F 78 41 00
```

Here the arrays pointed to by the `originalFirstThunk`'s have been highlighted in red, and those pointed to by the `firstThunk`'s in blue. They, namely the `firstThunk` arrays, have been grouped together in this way because the `firstThunk` arrays actually become the Import Address Table. The IAT needs to be in one solid, contiguous block which wouldn't be the case if we had grouped them by DLL instead.

And that's the final section finished! Let's go all the way back to the Data Directory in the Optional Header and update it. The size of the Import Table (the array of `IMAGE_IMPORT_DESCRIPTOR`'s, including the null entry) is `0x3C`; its RVA is `0x3000`. Let's fill in `size` and `virtualAddress` with these values:

```
0100 :00 30 00 00 3C 00 00 00 - 00 00 00 00 00 00 00 00
```

The IAT entry also needs to be filled in. The `virtualAddress` or RVA of the IAT is `0x304C` and its `size` is `0x10`.

```
0150 :00 00 00 00 00 00 00 00 - 4C 30 00 00 10 00 00 00
```

Fixing up `.text`

Now the `.idata` and `.rdata` sections are done, and we have their RVAs, we can fix up all those unknowns in our `.text` section. Let's have another look at it:

```
0200 :6A 40 68 XX XX XX XX 68 - XX XX XX XX 6A 00 FF 15
0210 :XX XX XX XX 6A 00 FF 15 - XX XX XX XX 00 00 00 00
```

The first two unknowns are the addresses of the string literals. The next two are the addresses of the function addresses (they are indirect calls). However, because the functions' parameters have to be addresses and the CPU expects an address for function calls, **none of these are RVAs** – they are **hardcoded**

addresses. So, once loaded into memory, where will our string literals be? Well, the `.rdata` section in which they reside is located at RVA `0x2000` – so if our executable is loaded at its preferred image base of `0x400000` (which it will be because it is an EXE and not a DLL), `.rdata` will be located at `0x400000 + 0x2000 = 0x402000`.

In the `.rdata` section we put `"Hello, World!",0` first and `"Message",0` afterwards. If we look back at this section we find the first at the very start of the section and the second at offset `0x0E` from the start of the section. So once loaded into memory, "Hello, World!" will be located at `0x402000` and "Message" at `0x40200E`. Substituting these values (remember, in our code we push the address of "message" before "Hello, World!"):

```
0200 :6A 40 68 0E 20 40 00 68 – 00 20 40 00 6A 00 FF 15
```

This leaves only the functions to fix up. Once the executable is loaded, the address of `MessageBoxA` will be at `0x403054` (the second non-null entry in the IAT) and the address of `ExitProcess` will be at `0x40304C`. Substituting these values, we get:

```
0200 :6A 40 68 0E 20 40 00 68 – 00 20 40 00 6A 00 FF 15
0210 :54 30 40 00 6A 00 FF 15 – 4C 30 40 00 00 00 00 00
```

Of course, if this was a DLL then there would be no guarantee that the executable would be loaded at `0x400000`, so we would need to add in a new section with a relocation table.

And that is it for the `.text` section. We are *almost* done! First, we need to fill out the `sizeOfImage` member in the Optional Header. `sizeOfImage` is the combined size of all headers and sections in memory, and is used by the OS to know how much memory in the process' address space to reserve for the executable image. We know our sections combined will take up 3 pages (`0x3000` bytes) but the headers are only `0x200` bytes long. However `sizeOfImage` must be a multiple of `sectionAlignment` so we round this up to `0x1000`, giving a final value of `0x3000 + 0x1000 = 0x4000`.

```
00D0 :00 40 00 00 00 02 00 00 – 00 00 00 00 02 00 00 00
```

Next, we must fill in `sizeOfCode`, `sizeOfInitData` and `sizeOfUninitData`. We only have one code section which is `0x200` bytes in size. We have two sections of initialised data, each being the same size as the code section, so their combined size is `0x400` bytes. We don't have any uninitialised data sections, so this value is `0`.

```
0090 :00 00 00 00 E0 00 03 01 – 0B 01 00 00 00 02 00 00
00A0 :00 04 00 00 00 00 00 00 – XX XX XX XX XX XX XX XX
00B0 :XX XX XX XX 00 00 40 00 – 00 10 00 00 00 02 00 00
```

The next missing field is `addrOfEntryPoint`, the RVA of the entry point of our program. Our code section is at RVA `0x1000` and execution begins right at the very start, so our `addrOfEntryPoint` is `0x1000`:

```
00A0 :00 04 00 00 00 00 00 00 – 00 10 00 00 XX XX XX XX
```

The last two missing fields are `baseOfCode` and `baseOfData`, the RVAs of our (first) code and data sections. Our first code section is `.text` with an RVA of `0x1000`, and our first data section is `.rdata` with an RVA of `0x2000`.

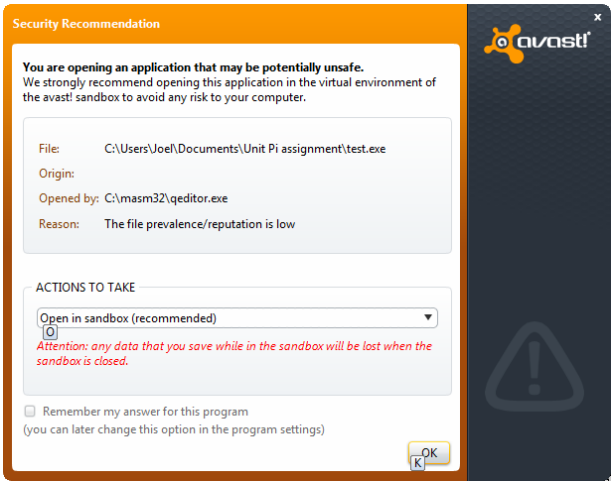
```
00A0 :00 04 00 00 00 00 00 00 – 00 10 00 00 00 10 00 00
00B0 :00 20 00 00 00 00 40 00 – 00 10 00 00 00 02 00 00
```

FINALLY our long and gruelling journey has come to a close – the PE file is ready to run (after conversion from hex of course). Here is the complete hex code (minus masses of zeroes) to check against:

```
0000 :4D 5A 77 00 01 00 00 00 - 04 00 00 00 00 00 00 00
0010 :B8 00 00 00 00 00 00 00 - 40 00 00 00 00 00 00 00
0020 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0030 :00 00 00 00 00 00 00 00 - 00 00 00 00 80 00 00 00     DOS HEADER + STUB PROGRAM
0040 :0E 1F B4 09 BA 0F 00 CD - 21 B4 4C B0 00 CD 21 54
0050 :68 69 73 20 70 72 6F 67 - 72 61 6D 20 63 61 6E 6E
0060 :6F 74 20 62 65 20 72 75 - 6E 20 69 6E 20 44 4F 53
0070 :20 6D 6F 64 65 2E 24 00 - 00 00 00 00 00 00 00 00
0080 :50 45 00 00 4C 01 03 00 - 00 00 00 00 00 00 00 00     PE SIGNATURE + COFF HEADER
0090 :00 00 00 00 E0 00 03 01 - 0B 01 00 00 00 02 00 00
00A0 :00 04 00 00 00 00 00 00 - 00 10 00 00 00 10 00 00
00B0 :00 20 00 00 00 00 40 00 - 00 10 00 00 00 02 00 00     OPTIONAL HEADER
00C0 :04 00 00 00 00 00 00 00 - 04 00 00 00 00 00 00 00
00D0 :00 40 00 00 00 02 00 00 - 00 00 00 00 02 00 00 00
00E0 :00 00 01 00 00 10 00 00 - 00 00 01 00 00 10 00 00
00F0 :00 00 00 00 10 00 00 00 - 00 00 00 00 00 00 00 00
0100 :00 30 00 00 3C 00 00 00 - 00 00 00 00 00 00 00 00
0110 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0120 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0130 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00     DATA DIRECTORY
0140 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0150 :00 00 00 00 00 00 00 00 - 4C 30 00 00 10 00 00 00
0160 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0170 :00 00 00 00 00 00 00 00 - 2E 74 65 78 74 00 00 00
0180 :1C 00 00 00 00 10 00 00 - 00 02 00 00 00 02 00 00
0190 :00 00 00 00 00 00 00 00 - 00 00 00 00 20 00 00 60
01A0 :2E 72 64 61 74 61 00 00 - 16 00 00 00 00 20 00 00
01B0 :00 02 00 00 00 04 00 00 - 00 00 00 00 00 00 00 00
01C0 :00 00 00 00 40 00 00 40 - 2E 69 64 61 74 61 00 00     SECTION TABLE
01D0 :90 00 00 00 00 30 00 00 - 00 02 00 00 00 06 00 00
01E0 :00 00 00 00 00 00 00 00 - 00 00 00 00 40 00 00 C0
01F0 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0200 :6A 40 68 0E 20 40 00 68 - 00 20 40 00 6A 00 FF 15
0210 :54 30 40 00 6A 00 FF 15 - 4C 30 40 00 00 00 00 00     .text: CODE SECTION
0220 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00

               <lots and lots of 00's>

03F0 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0400 :48 65 6C 6C 6F 2C 20 57 - 6F 72 6C 64 21 00 4D 65
0410 :73 73 61 67 65 00 00 00 - 00 00 00 00 00 00 00 00     .rdata: READ-ONLY DATA SECTION
0420 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00

               <lots and lots of 00's>

05F0 :00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
0600 :3C 30 00 00 00 00 00 00 - 00 00 00 00 5C 30 00 00
0610 :4C 30 00 00 44 30 00 00 - 00 00 00 00 00 00 00 00
0620 :69 30 00 00 54 30 00 00 - 00 00 00 00 00 00 00 00
0630 :00 00 00 00 00 00 00 00 - 00 00 00 00 74 30 00 00
0640 :00 00 00 00 82 30 00 00 - 00 00 00 00 74 30 00 00     .idata: IMPORT DATA SECTION
0650 :00 00 00 00 82 30 00 00 - 00 00 00 00 6B 65 72 6E
0660 :65 6C 33 32 2E 64 6C 6C - 00 75 73 65 72 33 32 2E
0670 :64 6C 6C 00 00 00 45 78 - 69 74 50 72 6F 63 65 73
0680 :73 00 00 00 4D 65 73 73 - 61 67 65 42 6F 78 41 00

       <lots and lots of 00's, till offset 0x800>
```

If you run the resulting program, you should get this:

But depending on the circumstances, you may get this instead:



Antivirus programs go crazy over small executables, and we've created the smallest executable possible given the section and file alignment limits (**in theory**, but it's possible to make even smaller ones...). However all ours does is display a message box and exit, so it's a false alarm.