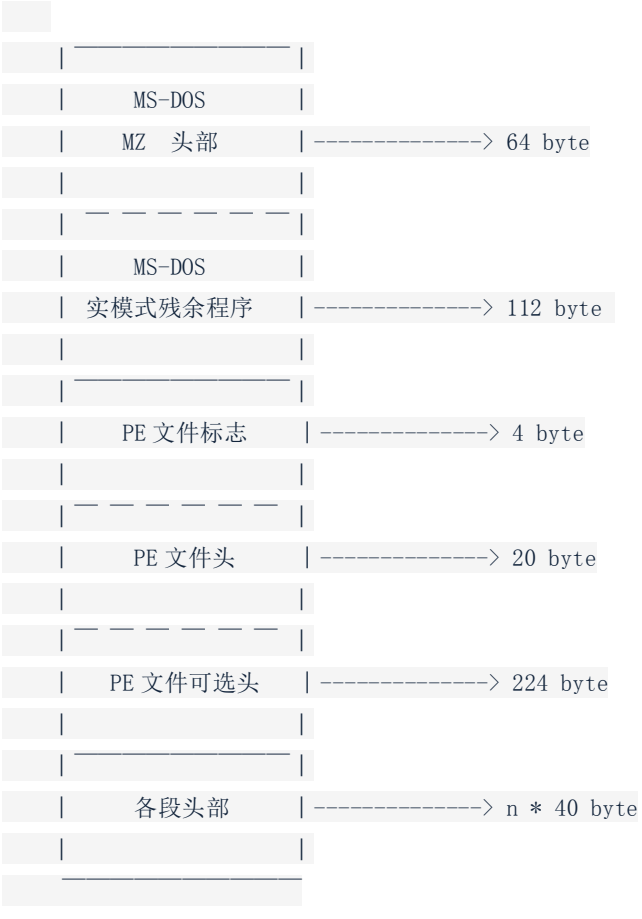
【文章标题】: 纯手工编写的PE可执行程序
【文章作者】: Kinney
【下载地址】: 自己搜索下载
【使用工具】: C32
【操作平台】: win 7
【作者声明】: 只是感兴趣，没有其他目的。失误之处敬请诸位大侠赐教!
最近，学习PE结构的知识。之后深有感触，随即便萌发了不依赖任何开发环境和编译器，纯手工写一个小程序的念头。所以我打算就写一个弹出MessageBox的小程序吧（弹出“Hello Kinney!This is the first PE program!”）。

在这里,我们首先复习一下Win32可执行程序的大体结构,就是通常所说的PE结构。PE 的意思就是Portable Executable（可移植的执行体）。

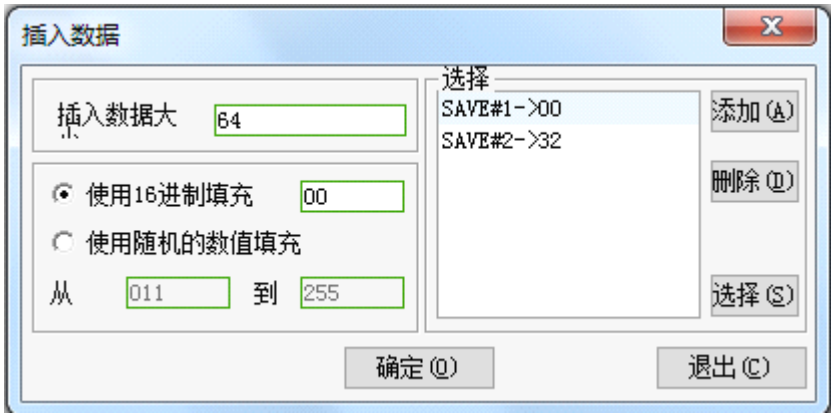
PE结构如下图：



IMAGE_DOS_HEADER:
每一个PE文件是以一个DOS程序开始的，有了它，一旦程序在DOS下执行，DOS就能识别出这是有效的执行体，然后运行紧随MZ Header之后的DOS stub（DOS残余块，其实是一个有效的EXE，就是我们可以看到的一个错误提示：This program cannot be run in MS-DOS mode)。我们对于这个DOS stub可以忽略，所以我在下面手写PE的时候，将DOS stub处填充为0。下面来看看IMAGE_DOS_HEADER的结构体定义，我将一些手写PE需要注意的几项给注释了下：

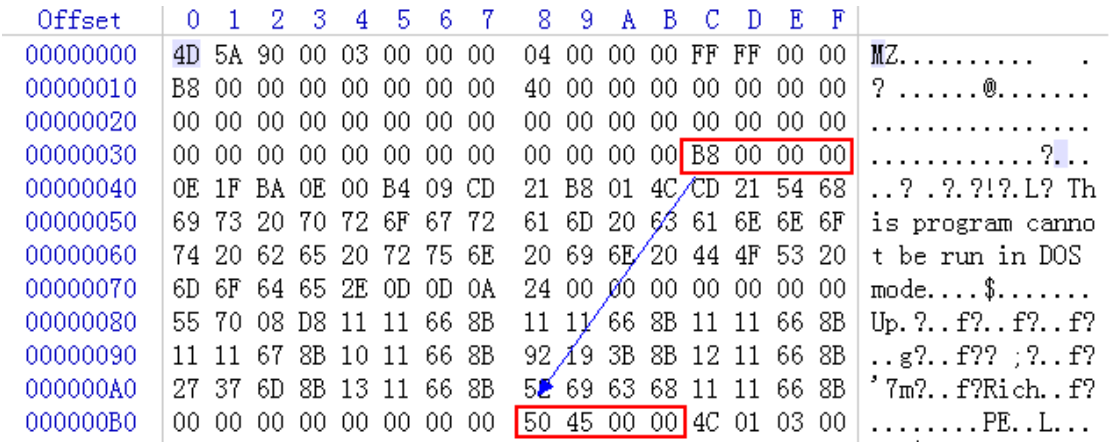
```
struct _IMAGE_DOS_HEADER {
0x00    WORD e_magic;         ; DOS 可执行文件标记“MZ”，被#define IMAGE_DOS_SIGNATURE 0x5A4Dh
0x02    WORD e_cblp;
0x04    WORD e_cp;
0x06    WORD e_crlc;
0x08    WORD e_cparhdr;
0x0a    WORD e_minalloc;
0x0c    WORD e_maxalloc;
0x0e    WORD e_ss;
0x10    WORD e_sp;
0x12    WORD e_csum;
0x14    WORD e_ip;            ; DOS 代码入口 IP
0x16    WORD e_cs;            ; DOS 代码的入口 CS
0x18    WORD e_lfarlc;
0x1a    WORD e_ovno;
0x1c    WORD e_res[4];
0x24    WORD e_oemid;
0x26    WORD e_oeminfo;
0x28    WORD e_res2[10];
0x3c    DWORD e_lfanew;       ; 指向 PE 文件头“PE”，0,0
};
```

MZ-DOS 头部占 64 个字节，所以我们在 C32 中选择插入 64 个 0：



图一

在 IMAGE_DOS_HEADER 中，有两个字段比较重要，分别是 e_magic 和 e_lfanew 字段（一个字大小）需要被设置为 5A4Dh，这个值是#define 的，在 ASCII 里，为“MZ”，是 MS-DOS 的最初创建者之一 Mark Zbikowski 字母的缩写，e_lfanew 字段是真正 PE 文件头的想对偏移（RVA），作用是指出真正 PE 头的文件偏移位置(如图二)：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..... .
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	? .....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B8	00	00	00	.....?...
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..? .?.?!?.L? Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.....
00000080	55	70	08	D8	11	11	66	8B	11	11	66	8B	11	11	66	8B	Up.?..f?..f?..f?
00000090	11	11	67	8B	10	11	66	8B	92	19	3B	8B	12	11	66	8B	..g?..f?? ;?..f?
000000A0	27	37	6D	8B	13	11	66	8B	52	69	63	68	11	11	66	8B	'7m?..f?Rich..f?
000000B0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	03	00	........PE..L...

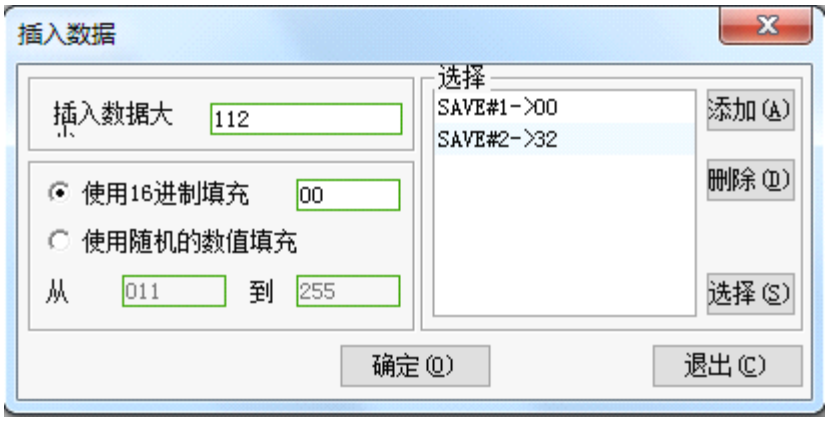
图二

从上面的结构体可以看出，它占 4 个字节，位于文件开始偏移 3Ch 字节中。 “PE 文件标志”紧随“MS-DOS 实模式残余程序”其后。知道这一点，我们就可以计算一下了，我们的“DOS MZ header”总共 64 byte，后面的“MS-DOS 实模式残余程序”占 112 byte， 64 + 112 = 176 byte,但是要注意，我们这里的 176 可是十进制的，转化成十六进制是 B0，对了，就是这个值，因为是 4 个字节，所以我们应该填“B0000000”。看上面的截图，为 B8000000，所以，保险起见，我们这里也填充为 B8000000.所以我们现在将前两个字节填充为 4D5A，在 3C 处填充为 B8000000。如图：

00000000:	4D	5A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	MZ.....
00000010:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000020:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030:	00	00	00	00	00	00	00	00	00	00	00	00	B8	00	00	00	.....?..

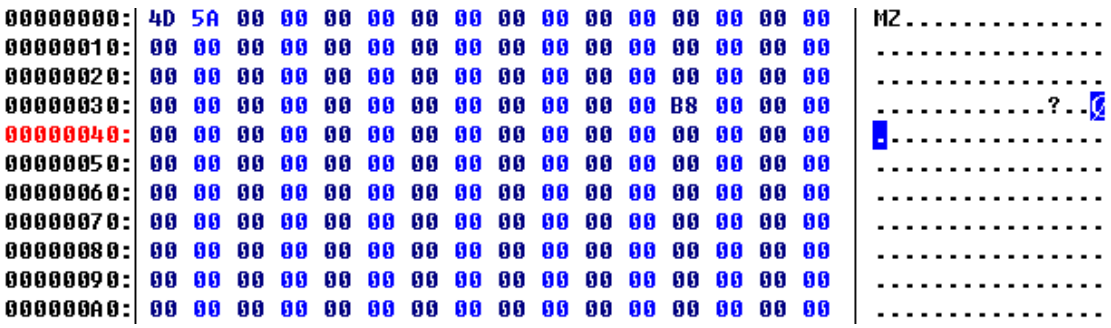
图三

接下来我们来完成“MS-DOS 实模式残余程序”，我们已经知道，他是用在 DOS 下执行的，我们这里可以直接用“00”来填充，注意总共 112 byte。 这两部分完成之后代码如下：



图四

填充好后，如图：



图五

在将准备工作做完以后，我们开始进入我们的重要部分，开始写真正的 PE 结构部分：

微软将“PE 文件标志”，“PE 文件头”，“PE 文件可选头”这三个部分用一个结构来定义，即：IMAGE_NT_HEADERS32（WINNT.H 中有定义，后面象这样的结构均在 WINNT.H 中有定义），

```
struct _IMAGE_NT_HEADERS {
0x00    DWORD Signature;                          ; PE 文件标识
0x04    _IMAGE_FILE_HEADER FileHeader;
0x18    _IMAGE_OPTIONAL_HEADER OptionalHeader;
};
```

这个结构含有 3 个成员：

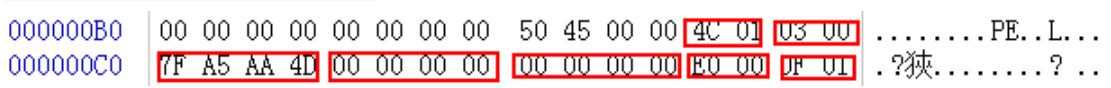
第一个成员表示“PE 文件标识”，可以看到他是一个 DWORD 类型，因此占 4 个字节，它是 PE 开始的标记，是一个#define IMAGE_NT_SIGNATURE 定义了这个值，对 Windows 程序这个值必须为“50450000”。DOS 头部的 e_lfanew 字段正是指向“PE\0\0”：

```
#define IMAGE_NT_SIGNATURE       0x00004550
```

第二个成员表示“PE 文件头 ”，他的类型是一个 IMAGE_FILE_HEADER 的结构。也就是说“PE 文件头”的 20 个字节被定义为 IMAGE_FILE_HEADER 结构，

```
struct _IMAGE_FILE_HEADER {
0x00    WORD Machine;                     ; 运行平台
0x02    WORD NumberOfSections;            ; 文件的区块数目
0x04    DWORD TimeDateStamp;              ; 文件创建日期和时间
0x08    DWORD PointerToSymbolTable;       ; 指向符号表（用于调试）
0x0c    DWORD NumberOfSymbols;            ; 符号表中符号个数（用于调试）
0x10    WORD SizeOfOptionalHeader;        ; IMAGE_OPTIONAL_HEADER32 结构的大小
0x12    WORD Characteristics;             ; 文件属性
};
```

这个结构具有 7 个成员（如图）：



成员 Machine，占 2 个字节，表示该文件运行所要求的 CPU。对于 Intel i386 平台，该值是“4C01”。

成员 NumberOfSections，占 2 个字节，表示该文件中段的总数，我们这里计划写 3 个段，（.text（代码段）、.rdata（只读数据段）、 .data（全局变量数据段））。所以此处值是“0300”。

成员 TimeDateStamp，占 4 个字节，表示文件创建日期和时间，从 1970.1.1 00:00:00 以来的秒数，我们这里填“0000”即可。

成员 PointerToSymbolTable，占 4 个字节，表示符号表的指针，主要用于调试，在这里填“0000”。

成员 NumberOfSymbols，占 4 个字节，表示符号的数目，主要用于调试，在这里填“0000”。

成员 SizeOfOptionalHeader，占 2 个字节，表示后面的“PE 文件可选头 ”部分所占空间大小，我们已经知道“PE 文件可选头 ”的大小是 224 byte，转换成十六进制就是 E0，所以这里的值为“E000”

成员 Characteristics，占 2 个字节，表示关于文件信息的标记，比如文件是 exe 还是 dll。这个值实际上是二进制位进行或运算得到的值。

各二进制位表示的意义如下：

Bit 0 ：置 1 表示文件中没有重定向信息。每个段都有它们自己的重定向信息。这个标志在可执行文件中没有使用，在可执行文件中是用一个叫做基址重定向目录表来表示重定向信息的，这将在下面介绍。

Bit 1 ：置 1 表示该文件是可执行文件（也就是说不是一个目标文件或库文件）。

Bit 2 ：置 1 表示没有行数信息；在可执行文件中没有使用。

Bit 3 ：置 1 表示没有局部符号信息；在可执行文件中没有使用。

Bit 4 ：

Bit 7

Bit 8 ：表示希望机器为 32 位机。这个值永远为 1。

Bit 9 ：表示没有调试信息，在可执行文件中没有使用。

Bit 10：置 1 表示该程序不能运行于可移动介质中（如软驱或 CD-ROM）。在这种情况下，OS 必须把文件拷贝到交换文件中执行。

Bit 11：置 1 表示程序不能在网上运行。在这种情况下，OS 必须把文件拷贝到交换文件中执行。

Bit 12：置 1 表示文件是一个系统文件例如驱动程序。在可执行文件中没有使用。

Bit 13：置 1 表示文件是一个动态链接库（DLL）。

Bit 14：表示文件被设计成不能运行于多处理器系统中。

Bit 15：表示文件的字节顺序如果不是机器所期望的，那么在读出之前要进行交换。在可执行文件中它们是不可信的（操作系统期望按正确的字节顺序执行程序）。

注意，因为我们写的是可执行程序，所以 Bit 1 必须置为 1，其他的按照需要置位即可，这里我们仅将第二位置位，由此得到成员 7 的值为“0200”。

第三个成员，表示“PE 文件可选头 ”，他的类型是一个 IMAGE_OPTIONAL_HEADER32 结构。也就是说“PE 文件头 ”的 224 个字节被定义为 IMAGE_OPTIONAL_HEADER32 结构，

```
struct _IMAGE_OPTIONAL_HEADER {
0x00    WORD Magic;                     ; 标志字
0x02    BYTE MajorLinkerVersion;        ; 链接器主版本号
0x03    BYTE MinorLinkerVersion;        ; 链接器次版本号
0x04    DWORD SizeOfCode;               ; 所有含有代码区块的总大小
0x08    DWORD SizeOfInitializedData;    ; 所有初始化数据区块总大小
0x0c    DWORD SizeOfUninitializedData;  ; 所有未初始化数据区块总大小
0x10    DWORD AddressOfEntryPoint;      ; 程序执行入口的 RVA
```

```
0x14   DWORD BaseOfCode;                    ；代码区块起始 RVA
0x18   DWORD BaseOfData;                    ；数据区块起始 RVA
0x1c   DWORD ImageBase;                     ；程序默认装入基地址
0x20   DWORD SectionAlignment;              ；内存中区块的对齐值
0x24   DWORD FileAlignment;                 ；文件中区块的对齐值
0x28   WORD MajorOperatingSystemVersion;    ；操作系统主版本号
0x2a   WORD MinorOperatingSystemVersion;    ；操作系统此版本号
0x2c   WORD MajorImageVersion;              ；用户自定义主版本号
0x2e   WORD MinorImageVersion;              ；用户自定义次版本号
0x30   WORD MajorSubsystemVersion;          ；所需要子系统主版本号
0x32   WORD MinorSubsystemVersion;          ；所需要子系统次版本号
0x34   DWORD Win32VersionValue;             ；保留，通常被设置为 0
0x38   DWORD SizeOfImage;                   ；影响装入内存后的总尺寸
0x3c   DWORD SizeOfHeaders;                 ；DOS 头、PE 头部、区块表总大小
0x40   DWORD CheckSum;                      ；影响校验和
0x44   WORD Subsystem;                      ；文件子系统
0x46   WORD DllCharacteristics;             ；显示 DLL 特性的旗标
0x48   DWORD SizeOfStackReserve;            ；初始化堆栈大小
0x4c   DWORD SizeOfStackCommit;             ；初始化实际提交堆栈大小
0x50   DWORD SizeOfHeapReserve;             ；初始化保留堆栈大小
0x54   DWORD SizeOfHeapCommit;              ；初始化实际保留堆栈大小
0x58   DWORD LoaderFlags;                   ；与调试有关，默认值为 0
0x5c   DWORD NumberOfRvaAndSizes;           ；数据目录表的项数
0x60   _IMAGE_DATA_DIRECTORY DataDirectory[16];
};
```

具有 31 个成员：

成员 1，占 2 个字节，表示文件的格式，值为 0x010B 表示.EXE 文件，为 0x0107 表示 ROM 映像，因为我们写的是一个可执行程序，所以此值应该为“0B01”。

成员 2，占 1 个字节，表示链接器的主版本号，此值不会影响程序的执行，我们这里填充零，此值为“00”。

成员 3，占 1 个字节，表示链接器的幅版本号，此值不会影响程序的执行，我们这里填充零，此值为“00”。

成员 4，占 4 个字节，表示可执行代码的长度，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 5，占 4 个字节，表示初始化数据的长度（数据段）。此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 6，占 4 个字节，表示未初始化数据的长度（bss 段）。此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

（在介绍成员 7 之前，有必要了解一个很重要的知识------文件映射到内存。在可执行程序运行之前，PE 加载器将把 PE 文件加载到进程空间的内存中去，并且初始化每个段实体。那么加载到内存中的哪个地址去呢？这将由 IMAGE_OPTIONAL_HEADER32 结构的成员 10 的值指出加载的起始地址（又叫基地址）。

这个值通常是“00400000”， 那么 PE 文件的首地址“00000”就被映射到内存地址“00400000”处，那么相对于文件偏移 10 个字节的地址为“00010”，被映射到内存后的偏移也应该是 10 个字节，映射后的地址应该为“00400010”。）

成员 7，4 个字节，表示代码的入口 RVA（文件映射到内存的偏移地址）地址，程序从这儿开始执行。PE 装载器准备运行的 PE 文件的第一个指令的 RVA。若您要改变整个执行的流程，可以将该值指定到新的 RVA，这样新 RVA 处的指令首先被执行。那么这个值我们怎么得到呢？我们知道在文件中有个.text 段，他包含了所有的代码，我们可以从中找到我们的入口地址，在这里就是.text 段里的第一行代码，也就是.text 段的首地址，而在.text 段头部就给出了他映射到内存后的首地址的偏移，我们找到他取出添到此处，这里为“00100000”。（此处不理解没关系，我们讲完段结构后自能迎刃而解。）

成员 8，4 个字节，表示可执行代码起始位置。当然就是.text 段的首地址，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 9，4 个字节，表示初始化数据的起始位置，此值不会影响程序的执行，我们这里填充零，此值为“00000000”。

成员 10，4 个字节，就是上面所讲的文件映射到内存是的基地址。PE 文件的优先装载地址。通常设为“00400000”，PE 装载器将尝试把文件装到虚拟地址空间的 00400000h 处。字眼"优先"表示若该地址区域已被其他模块占用，那 PE 装载器会选用其他空闲地址。我们这里的值设为“00400000”。

成员 11，4 个字节，表示段加载后在内存中的对齐方式。内存中节对齐的粒度。例如，如果该值是 4096 (1000h)，那么每节的起始地址必须是 4096 的倍数。若第一节从 401000h 开始且大小是 10 个字节，则下一节必定从 402000h 开始，即使 401000h 和 402000h 之间还有很多空间没被使用。因为 Windows 管理内存采用分页管理的方式，而每页的大小为 4k，也就是 1000h，所以我们这个值为“00100000”。

成员 12，4 个字节，表示段在文件中的对齐方式。文件中节对齐的粒度。例如，如果该值是(200h)，，那么每节的起始地址必须是 512 的倍数。若第一节从文件偏移量 200h 开始且大小是 10 个字节，则下一节必定位于偏移量 400h: 即使偏移量 512 和 1024 之间还有很多空间没被使用。此值最好设为 200h，所以该成员的值为“00020000”。

成员 13，2 个字节，表示操作系统主版本号，此值不会影响程序的执行，我们这里填充零，此值为“0000”。

成员 14，2 个字节，表示操作系统副版本号，此值不会影响程序的执行，我们这里填充零，此值为“0000”。

成员 15，2 个字节，表示程序主版本号，此值不会影响程序的执行，我们这里填充零，此值为“0000”。

成员 16，2 个字节，表示程序副版本号，此值不会影响程序的执行，我们这里填充零，此值为“0000”。

成员 17，2 个字节，表示子系统主版本号。win32 子系统版本。PE 文件是专门为 Win32 设计的，该子系统版本必定是 4.0 那么此处值为“04”。

成员 18，2 个字节，表示子系统副版本号，根据上面所说，此值应为“00”。

成员 19，2 个字节，此值一般为“00”。

成员 20，4 个字节，表示程序调入后占用内存大小（字节），等于所有段的长度之和。所有头和节经过节对齐处理后的大小。我们知道，我们文件 PE 结构总长小于 1000h，但是内存中的对齐粒度是 1000h，所以 PE 结构被映射后要占 1000h，尽管很多空间没有使用，另外我们有 3 个段，每个段的长度小于 1000h，但是被映射后同样要占 1000h，所以总共占用内存的大小为 1000h + 3 * 1000h = 4000h，因此此值为“00400000”。

成员 21，4 个字节，表示所有文件头的长度之和(从文件开始到第一个段之间的大小)。所有头+节表的大小，也就等于文件尺寸减去文件中所有节的尺寸。可以以此值作为 PE 文件第一节的文件偏移量。那么我们怎么得到这个值呢？我们的 PE 文件结构总大小为：64 + 112 + 4 + 20 + 224 + 3 * 40 = 544 byte

转化成十六进制为220h，那么此值就是220h吗？

不是的，因为我们文件中的对齐粒度是200h，那么220h实际上要占用400h的空间，所以此值为“00040000”。

成员22，4个字节，表示校验和。它仅用在驱动程序中，在可执行文件中可能为0。它的计算方法Microsoft不公开，在imagehelp.dll中的CheckSumMappedFile()函数可以计算它，此处我们设为填充零，此值为“00000000”。

成员23，2个字节，表示NT子系统，可能是以下的值：

IMAGE_SUBSYSTEM_NATIVE (1) 不需要子系统。用在驱动程序中。

IMAGE_SUBSYSTEM_WINDOWS_GUI(2) WIN32 graphical程序（它可用AllocConsole()来打开一个控制台，但是不能在一开始自动得到）。

IMAGE_SUBSYSTEM_WINDOWS_CUI(3) WIN32 console程序（它可以一开始自动建立）。

IMAGE_SUBSYSTEM_OS2_CUI(5) OS/2 console程序（因为程序是OS/2格式，所以它很少用在PE）。

IMAGE_SUBSYSTEM_POSIX_CUI(7) POSIX console程序。

Windows程序总是用WIN32子系统，所以只有2和3是合法的值。也就是说此值必须为2或3，如果是3，那么程序运行后会自动打开一个控制台，我们为了看一下效果，这里设为3，此值为“0300“。

成员24，2个字节，表示Dll状态，我们这里填充零，此值为“0000”。

成员25，4个字节，保留堆栈大小，我们这里填充零，此值为“00000000”。

成员26，4个字节，启动后实际申请的堆栈数，可随实际情况变大，我们这里填充零，此值为“00000000”。

成员27，4个字节，保留堆大小，我们这里填充零，此值为“00000000”。

成员28，4个字节，实际堆大小，我们这里填充零，此值为“00000000”。

成员29，4个字节，装载标志，我们这里填充零，此值为“00000000”。

成员30，4个字节，在讲这个成员之前，我们应该先了解成员31，成员31实际上是一个IMAGE_DATA_DIRECTORY结构的数组，成员30的值就是表示该数组的大小。通常有16个元素，所以此值为:“10000000”。

```
IMAGE_DIRECTORY_ENTRY_EXPORT
struct _IMAGE_DATA_DIRECTORY {
0x00   DWORD VirtualAddress;
0x04   DWORD Size;
};
```

成员31，128个字节，上面说过他是一个IMAGE_DATA_DIRECTORY结构的数组，通常具有16个元素。

IMAGE_DATA_DIRECTORY结构有两个成员，各占4个字节，那么也就得到成员31的总大小：2＊4＊16=128byte。

中每个元素代表一个目录表，每个目录表表示的目录如下：

IMAGE_DIRECTORY_ENTRY_EXPORT (0)导出目录用于DLL

IMAGE_DIRECTORY_ENTRY_IMPORT (1导入目录

IMAGE_DIRECTORY_ENTRY_RESOURCE (2)资源目录

IMAGE_DIRECTORY_ENTRY_EXCEPTION (3)异常目录

IMAGE_DIRECTORY_ENTRY_SECURITY (4)安全目录

IMAGE_DIRECTORY_ENTRY_BASERELOC (5)重定位表

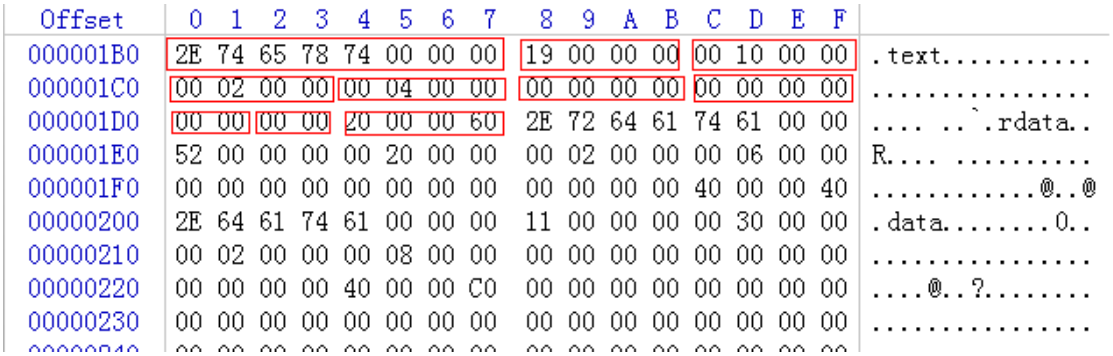
IMAGE_DIRECTORY_ENTRY_DEBUG (6)调试目录
IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7)描述版权串
IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8)机器值
IMAGE_DIRECTORY_ENTRY_TLS (9)Thread local storage目录
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10)Load configuration 目录
IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11)Bound import directory目录
IMAGE_DIRECTORY_ENTRY_IAT (12)Import Address Table输入地址表目录
IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT 13 // Delay Load Import Descriptors
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14 // COM Runtime descriptor

是不是所有的目录表都要关心呢？其实要把这些目录表都研究清楚是个很大的课题，对于我们这个程序，只需关心第2个元素，导入目录，它标识了我们的程序从其他模块导入的函数信息。因为我们要显示一个消息框，所以要导入user32.dll库中的MessageBoxA函数，程序退出，又要导入kernel32.dll库中的ExitProcess函数，这个目录表需要使用。然而上面已说明每个目录是一个IMAGE_DATA_DIRECTORY结构，该结构具有两个成员，第一个成员表示目录表的起始RVA地址，第二个成员表示目录表的长度。这两个值要根据.rdata段实体来确定，暂时先不填写。为了记录该位置，我们先都填写为x，即：
“xxxxxxxx","xxxxxxxx"。其余的统统添零即可。

接下来是各段头部，我们这里有3个段，.text(代码段)， .rdata(只读数据段)，data(全局变量数据段)。每段是一个IMAGE_SECTION_HEADER 结构，具有10个成员。首先我们来看.text段。

```
typedef struct _IMAGE_SECTION_HEADER {
0x00   BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
union {
0x08  DWORD PhysicalAddress;
0x08 DWORD VirtualSize;
} Misc;
0x0c   DWORD VirtualAddress;          ；区块的RVA地址
0x10   DWORD SizeOfRawData;           ；在文件中对齐后的尺寸
0x14   DWORD PointerToRawData;        ；在文件中偏移
0x18   DWORD PointerToRelocations;    ；在OBJ文件中使用，重定位的偏移
0x1c   DWORD PointerToLinenumbers;    ；行号表的便宜（供调试用）
0x20   WORD  NumberOfRelocations;     ；在OBJ文件中使用，重定位项数目
0x22   WORD  NumberOfLinenumbers;     ；行号表中行号的数目
0x24   DWORD Characteristics;         ；区块的属性
};
```

如图，各个成员分布如图：



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000001B0	2E	74	65	78	74	00	00	00	19	00	00	00	00	10	00	00	.text..........
000001C0	00	02	00	00	00	04	00	00	00	00	00	00	00	00	00	00	....
000001D0	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00	.... ..`.rdata..
000001E0	52	00	00	00	00	20	00	00	00	02	00	00	00	06	00	00	R.... ....
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	............@..@
00000200	2E	64	61	74	61	00	00	00	11	00	00	00	00	30	00	00	.data........0..
00000210	00	02	00	00	00	08	00	00	00	00	00	00	00	00	00	00	....
00000220	00	00	00	00	40	00	00	C0	00	00	00	00	00	00	00	00	....@..?........
00000230	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

成员 1，8 个字节，表识该段的名称，我们这里是.text，那么此值是他的 ASCII 码应该为“2E7465787400000”。

成员 2，4 个字节，表示有效代码所占的字节数。我们这里所有代码数一下总共 26h 个，固此值为“26000000”。

成员 3，4 个字节，表示在.text 段映射到内存中的起始地址，那么这个值如何得来呢？我们知道.text 是紧跟 PE 结构后的，然后整个 PE 结构映射到内存后占的大小为 1000h（因为 PE 结构小于 1000h 个字节，而对齐力度粒度是 1000h），那么此值便得到了，为“00100000”。

成员 4，4 个字节，表示.text 段在文件中所占的大小。因为我们的实际代码只有 26h 个字节，那么这个值是不是 26h 呢？并不是，一定要注意段在文件中的对齐粒度是 200h，所以此值为“00020000”。

成员 5，4 个字节，表示.text 段在文件中的起始地址，上面已经计算过 PE 文件的总长度为 400h，他实际上也就是.text 的起始偏移地址，此值为“00040000”。

成员 6，7，8，9，均占 4 个字节，都仅用于目标文件，我们这里统统填为零。

成员 10，4 个字节。包含标记以指示节属性，比如节是否含有可执行代码、初始化数据、未初始数据，是否可写、可读等。这个值实际上是二进制位进行或运算得到的值。各二进制位表示的意义如下：

bit 5 (IMAGE_SCN_CNT_CODE)，置 1，节内包含可执行代码。

bit 6 (IMAGE_SCN_CNT_INITIALIZED_DATA)置 1，节内包含的数据在执行前是确定的。

bit 7 (IMAGE_SCN_CNT_UNINITIALIZED_DATA) 置 1，本节包含未初始化的数据，执行前即将被初始化为 0。一般是 BSS.

bit 9 (IMAGE_SCN_LNK_INFO) 置 1，节内不包含映象数据除了注释，描述或者其他文档外，是一个目标文件的一部分，可能是针对链接器的信息。比如哪个库被需要。

bit 11 (IMAGE_SCN_LNK_REMOVE) 置 1，在可执行文件链接后，作为文件一部分的数据被清除。

bit 12 (IMAGE_SCN_LNK_COMDAT) 置 1，节包含公共块数据，是某个顺序的打包的函数。

bit 15 (IMAGE_SCN_MEM_FARDATA) 置 1，不确定。

bit 17 (IMAGE_SCN_MEM_PURGEABLE) 置 1，节的数据是可清除的。

bit 18 (IMAGE_SCN_MEM_LOCKED) 置 1，节不可以在内存内移动。

bit 19 (IMAGE_SCN_MEM_PRELOAD)置 1， 节必须在执行开始前调入。

Bits 20 to 23 指定对齐。一般是库文件的对象对齐。

bit 24 (IMAGE_SCN_LNK_NRELOC_OVFL) 置 1， 节包含扩展的重定位。

bit 25 (IMAGE_SCN_MEM_DISCARDABLE) 置 1，进程开始后节的数据不再需要。

bit 26 (IMAGE_SCN_MEM_NOT_CACHED) 置 1，节的 数据不得缓存。

bit 27 (IMAGE_SCN_MEM_NOT_PAGED) 置 1，节的 数据不得交换出去。

bit 28 (IMAGE_SCN_MEM_SHARED) 置 1，节的数据在所有映象例程内共享，如 DLL 的初始化数据。

bit 29 (IMAGE_SCN_MEM_EXECUTE) 置 1，进程得到“执行”访问节内存。

bit 30 (IMAGE_SCN_MEM_READ) 置 1，进程得到“读出”访问节内存。

bit 31 (IMAGE_SCN_MEM_WRITE)置 1，进程得到“写入”访问节内存。

在我们这里，因为这是代码段，所以 bit 5 (IMAGE_SCN_CNT_CODE)位置 1，一般代码段都含有初始化数据，那么 bit 6 (IMAGE_SCN_CNT_INITIALIZED_DATA)位置 1，有因为代码段的代码可以执行的，所以 bit 29 (IMAGE_SCN_MEM_EXECUTE) 置 1,那么这 3 个二进制位进行或运算最终得到此成员值“20000060”。

这个整个.text 头就编写完毕，按照上面的方法，分别在编写.rdata 段和.data 段。因为要对齐，所以后面的代码用零补齐。

最后的编写结果如下：

```
000000B0: 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00 ........PE..L...
000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 0F 01 ............?...
000000D0: 0B 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000E0: 00 10 00 00 00 00 00 00 00 00 00 00 00 00 40 00 ..............@.
000000F0: 00 10 00 00 00 02 00 00 00 00 00 00 00 00 00 00 ................
00000100: 04 00 00 00 00 00 00 00 00 40 00 00 00 04 00 00 .........@......
00000110: 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ................
00000120: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 ................
00000130: 00 00 00 00 00 00 00 00 08 20 00 00 28 00 00 00 ......... ..(...
00000140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001B0: 2E 74 65 78 74 00 00 00 16 00 00 00 00 10 00 00 .text............
000001C0: 00 02 00 00 00 04 00 00 00 00 00 00 00 00 00 00 ................
000001D0: 00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00 .... ..`.rdata..
000001E0: 52 00 00 00 00 20 00 00 00 02 00 00 00 06 00 00 R.... ..........
000001F0: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 ............@..@
00000200: 2E 64 61 74 61 00 00 00 04 00 00 00 00 30 00 00 .data........0..
00000210: 00 02 00 00 00 08 00 00 00 00 00 00 00 00 00 00 ................
00000220: 00 00 00 00 40 00 00 C0 00 00 00 00 00 00 00 00 ....@..?........
00000230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
00000380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000003F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

至此，我们已经完成了 PE 结构的编写。但是，此时的程序还不能够运行，我们还需要再耐心的补上点东西。

为了让我们写的程序可以运行，我们还要完成.text(代码段)， .rdata(只读数据段)，data(全局变量数据段)三个段的实体部分。

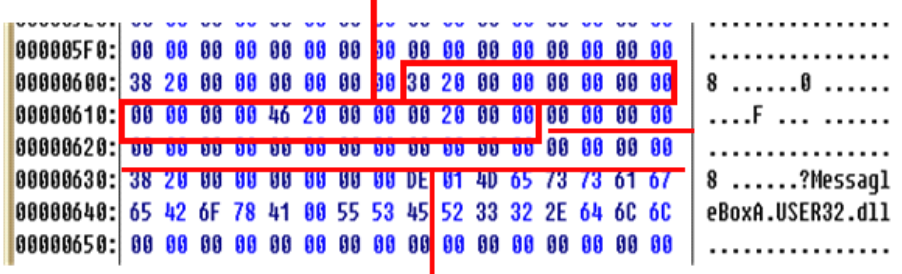
首先编写.text 段，他紧接着 PE 结构后面，但是我们如何编写这些内容呢？前面已经说过，.text 段中存放所有的可执行代码（机器码），我们可以通过先编写汇编指令（调用 MessageBoxA 和 ExitProcess 两个函数），然后反汇编出机器代码抄到这里就可以了。这里有一点要注意，我们在为 MessageBoxA 函数传递参数时，如何将“Hello Kinney!This is the first PE program!”字符串这就要用到我们的.data(全局变量数据段)了，我们可以把这两个字符串放到这个段中，然后把字符串的偏移首地址作为参数传给 MessageBoxA 即可。因为要以 200h 对齐，所以剩余部分用零补齐，最终得到的代码如下：

```
00000400: B8 00 30 40 00 6A 00 50 50 6A 00 FF 15 00 20 40 .0@.j.PPj.ÿ.. @
00000410: 00 33 C0 C2 10 00 00 00 00 00 00 00 00 00 00 00 .3缇............
00000420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000490: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000004F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000510: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000520: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000530: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000540: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000590: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000005F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

接下来完成.rdata 段，这个段非常重要，也有些繁琐。要写入导入表（_IMAGE_IMPORT_DESCRIPTOR），相当的繁琐，我们找一个 C++编译过的程序，载入 C32 中，分析下它的导入表，如图：

这里就是IMAGE_IMPORT_DESCRIPTOR结构共20字节

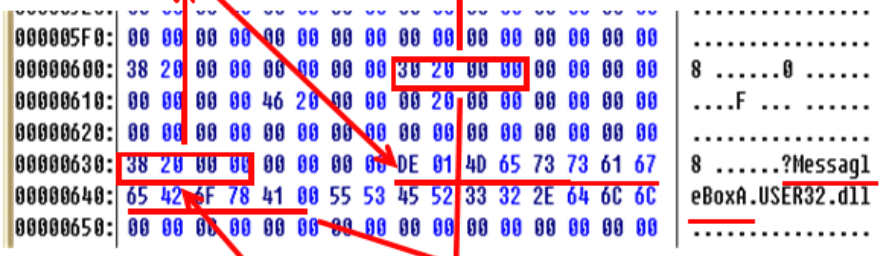
我举的例子里只有一个dll所以只需要一个IMAGE_IMPORT_DESCRIPTOR结构，如果还需要导入其他的dll那么就要紧随其后继续写入这个结构，最后用一个全0的IMAGE_IMPORT_DESCRIPTOR结构结尾。



这里就是全0结尾的IMAGE_IMPORT_DESCRIPTOR结构共20字节

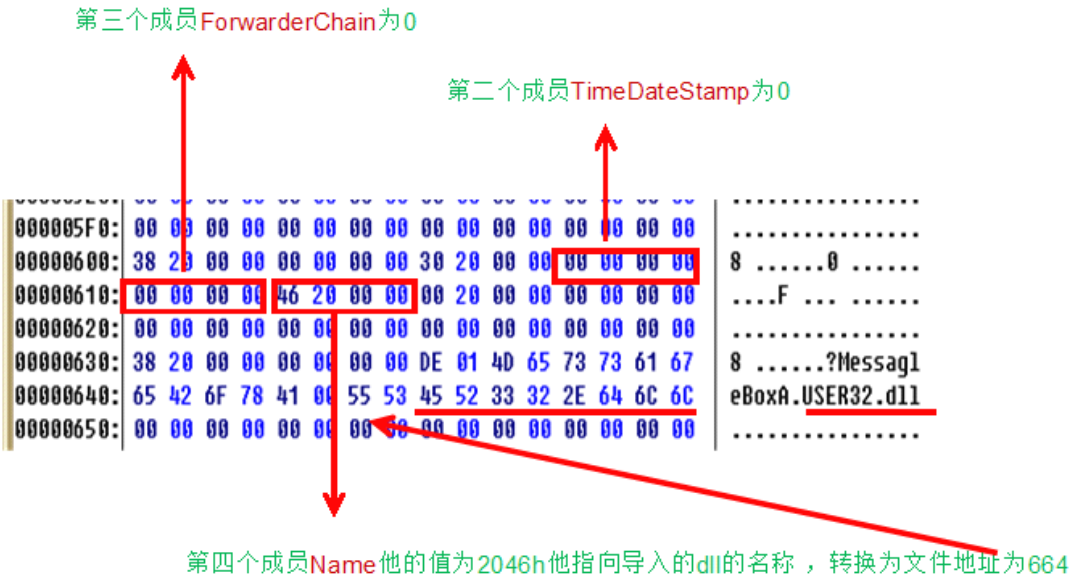
第一个成员OriginalFirstThunk他的值为2030h他指向IMAGE_THUNK_DATA 。

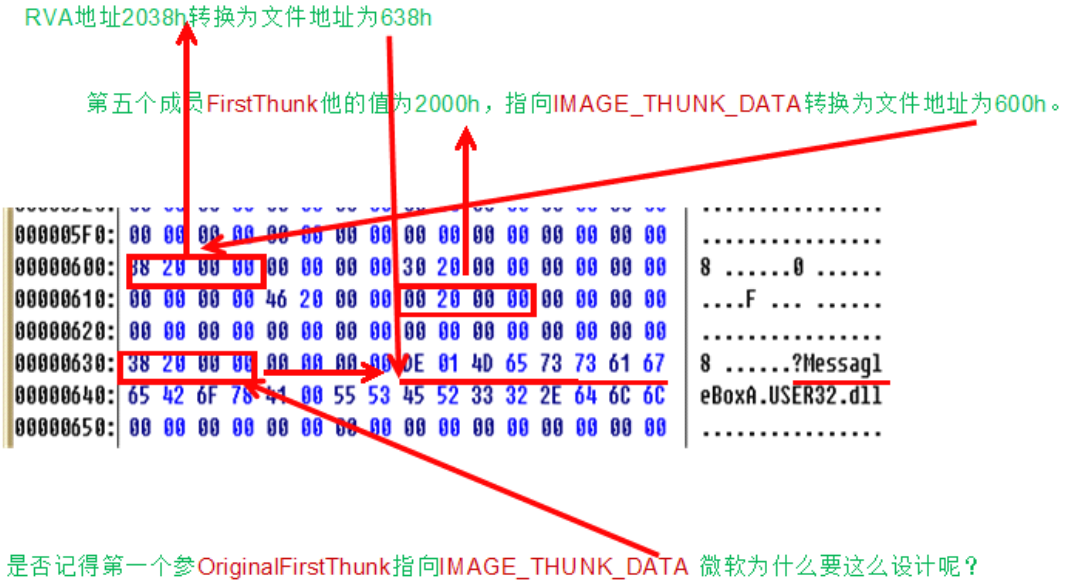
RVA地址2038h转换为文件地址为638h



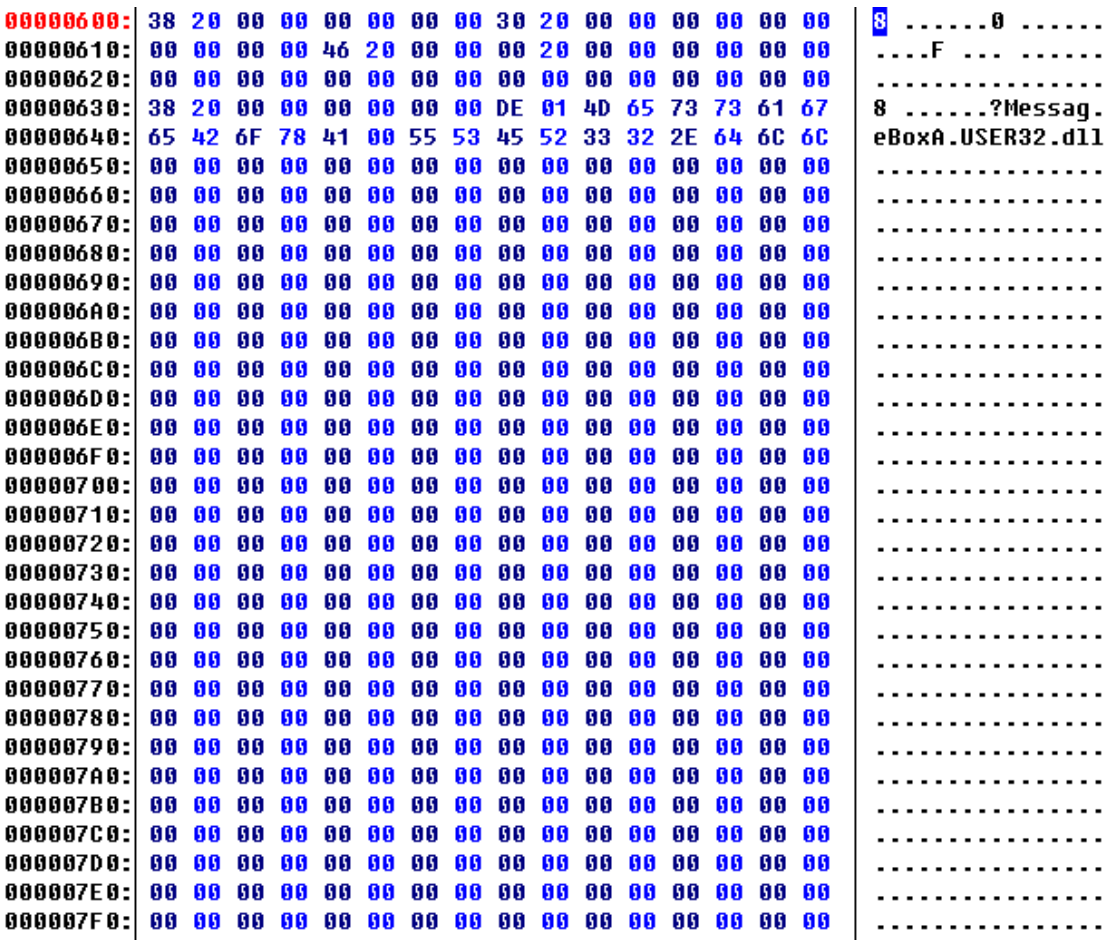
导入的API的名字"MessageBoxA"，注意这里并不是正好的MessageBoxA，前面还多出一个数据，今天我们不去深究这里知道就可以了。

用lordpe计算一下他的RVA地址2030h转换为文件地址为630h





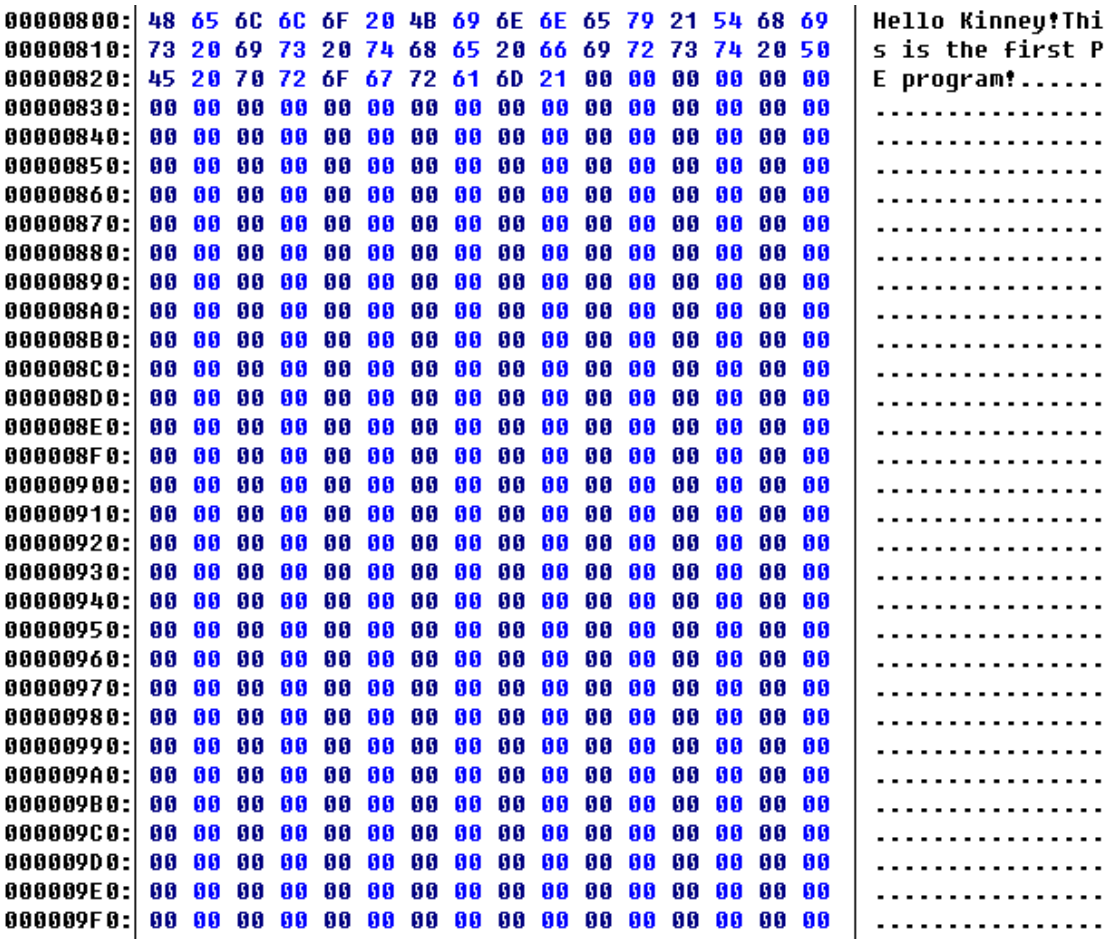
由此，我们可以完成我们的.rdata 段的写入了，写好的代码如下：



```
00000600: 38 20 00 00 00 00 00 00 30 20 00 00 00 00 00 00 | 8 ......0 ......
00000610: 00 00 00 00 46 20 00 00 00 20 00 00 00 00 00 00 | ....F ... ......
00000620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000630: 38 20 00 00 00 00 00 00 DE 01 4D 65 73 73 61 67 | 8 ......?Messag.
00000640: 65 42 6F 78 41 00 55 53 45 52 33 32 2E 64 6C 6C | eBoxA.USER32.dll
00000650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000660: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000690: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000006F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000710: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000720: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000730: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000740: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000750: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000760: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000770: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
00000790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
000007F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
```

最后一个是.data 段，这个段非常简单，就是 MessageBoxA 所需的参数，消息框的内容，最终代码如下：

（注意对齐问题，补足 200h 字节。）



好了，到此为止一个完整的显示 Hello Kinney!This is the first PE program!的可执行程序就完成了，木有用到编译器和链接器等等，爽吧？但是很累的说~！赶快双击运行一下吧...费了这么大劲就这么个小功能，是不是有点事倍功半呢？其实手写这么个程序只是为了更加熟练掌握 PE 结构，相信当您真正的手工完成了这个程序，您对 PE 结构一定有一个非常深刻的理解。（提示：请按照以上步骤完成这个程序，之后再回头从头到尾联系上下文仔细看一遍，因为很多地方都是前后关联紧密的，只作一遍，或只读一遍是很难融汇贯通的。）

我手写的 PE 文件会打包上去，由于水平和时间的原因，难免有许多问题，或者表述不清楚的地方，请高手不吝赐教。
