Operating Systems Development Series

Operating Systems Development - Portable Executable (PE)

by Mike, 2011

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome!

Yey, this is going to be a long one.

This chapter is going to cover an advanced topic - the PE executable file format. We will be looking at covering PE resources, dynamic linking, and more. This chapter is also planned for an update to include more information to make the information as complete as possible.

Most of what is included in this chapter is for information purposes only and are only included both for completness and in case any of our readers would like to provide support for them. Also please note that a lot of the information provided can also be found in the official PE specification.

After this chapter, we will have everything we need to develop a loader and support a single tasking environment.

Lets get started!

File Format

Abstract

The **Portable Executable (PE)** file format is the standard executable file format used in several operating systems, including Windows and Windows-like OSs, such as ReactOS. It is also the standard file format used with booting on **Extensable Firmware Interface (EFI)** machines.

The PE executable file format is a complex format, supporting relocations, symbol tables, resources, dynamic binding, and more.

Terms

VA (Virtual Address)

A **Virtual Address (VA)** is a linear address in the **Virtual Address Space (VAS)** of the current program. All addresses in the PE executable format are virtual addresses. These addresses are 32 bit linear addresses.

RVA (Relative Virtual Address)

A **Relative Virtual Address (RVA)** is a VA that is relative to the **base address** of the executable program. The PE executable format uses RVAs in a lot of areas, so it is important to know what RVAs are and how to obtain linear addresses from them. RVAs are just offsets from the base address, thats all. So to obtain its linear address, just add the RVA to the base:

```
Linear address = Base address + RVA
```

This one is important as we will need to perform this calculation in a lot of areas when parsing.

Sections and the Section Table

Sections

Advanced executable file formats typically use **program sections** to simplify the linking process and provide structure to the software. Sections simplifies the linking process by providing a standard method for instructions and data to be stored within the executable image or object file.

A section typically has a name associated with what elements are inside of the section. For example, **.data** is a common section name that contains variable, uninitialized data. Other section names have historical backgrounds. For example, **.text** is a typical name of a section containing executable or object code. **.bss** is typically used for global, program-wide initialized data.

Using the C++ toolchain, for an example, variables defined in the global namespace or as **static** are stored in **.bss**. The resulting bytecode generated after compilation is stored in **.text**.

The PE executable file format typically includes one or more of the following section names:

- .text
- .data
- .bss
- .arch
- .edata
- .idata
- .pdata
- .rdata
- .reloc
- .rsrc
- .sbss
- .sdata
- .srdata
- .xdata

Section Table

Program files and object files contain multiple sections. The base location of each section, and the name of that section, is typically stored in a **section table**. In some implimentations, a section table can be a simple linked list of structures or a hash table - different implimentations exist.

Symbols and the Symbol Table

Symbols

While programming in C++, you most likley have encountered the unfamous **Undefined symbol** linker errors (or in some implimentations of C, warnings. Thats right, fully compiling and linking without error *ahem* old MSVC). This happens do to calling a function or referencing a variable by name whose definition could not be resolved during the linking stage.

Functions and variables are referred to as **symbols** by the linker. A **symbol** contains the name, and information about what it is: such as a data type and value, for example. During compiling, the compilier must keep track of these symbols to insure that the final program can be linked. If a symbol is used that is not defined in the current **translation unit**, but is an EXTERN symbol, the compilier will need to mark it as an EXTERN symbol when writing the symbols to the object file.

If, during the linking stage, a symbol marked EXTERN still has no value associated with it (the symbol has no definition), the linker issues the above error.

Symbols are what allows programmers to define variables or functions across modules, translation units, or libraries.

There can only be one symbol with the same name through the entire program, and any libraries that it links with. Because of this, and the high probability of naming collisions with the use of high level languages, variable and function names typically have **name mangling** applied. This, of course, does not apply to assembly language. The name mangling applied depends on multiple factors, and differ between toolchains.

Lets take a look. Here are some C function declarations, and their mangled symbolic name on the right. The number in the mangled name is the number of bytes for the paramaters.

```
void _cdecl function (int i);   ->   _function
void _stdcall function(int i);  ->   _function@4
void _fastcall function(int i); ->   @function@4
```

Notice that functions with the _cdecl call convention only have an underscore prepended. This allows C functions to be easily defined using assembly language, and allows C code to easily call those functions.

There isnt any standard for C++ name mangling. Some compiliers might produce a symbolic name like **?h@@YAXH@Z**, while others can produce **__7h__Fi** or **W?h$n(i)v** for the **same** function of **void h(int)**. This makes it impractical to use with assembly language. It is still possible, however.

Symbol table

Simular to the section table there exists a **symbol table**. The symbol table allows a way for the software to look up symbolic names and information about the symbol, such as if its an exported symbol, its data type, properties, etc. Symbol tables are typically a linked list of information or implimented in hash tables.

Structure

Abstract

We have taken a look at the structure of the PE executable format in our MSVC++ chapter. When we load a PE executable in memory, that memory would contain an exact copy of our loaded file. This means the first byte within the first structure of the PE file format is actually located at the first byte from where the file was loaded in memory.

For example, if we load a PE file to 1MB, the in-memory footprint might look like this:

Notice that the file contents directly start at where it was loaded in memory. This means, in order to read file contents, read it from the loaded image file.

The above image should look famailier to our readers who have read the MSVC chapter. Looking at the above image, if the PE file was loaded to 1MB, then the first on-disk structure, IMAGE_DOS_HEADER, begins at that location in memory, followed by the rest of the structures in the file (including padding).

The above image is also an oversimplification - it does not, by any means, show the complete PE file format. The structure of the PE file format is fairly large, composed of a lot of structures and tables.

Here is the complete format:

1. IMAGE_DOS_HEADER structure **(Important)**
2. STUB program
3. IMAGE_FILE_HEADER structure [COFF Header] **(Important)**
4. IMAGE_OPTIONAL_HEADER structure **(Important)**
5. Segment Table
6. Resource Table
7. Resident Name Table
8. Module Reference Table
9. Imported Names Table
10. Entry Table
11. Non Resident Name Table
12. Segments
    1. Data
    2. Info

The above table lists the complete file format - from beginning to end. Items marked **important** are required to know how to parse in order to just execute the program. All of the other information if provided for information purposes only. The only important members in **IMAGE_OPTIONAL_HEADER** structure are the member that contains the address of the entry point, and image base address.

We will cover parsing each section of this file in detail in the upcoming sections. We will also introduce the other structures used when parsing tables and directories as well.

IMAGE_DOS_HEADER structure

IMAGE_DOS_HEADER is the first structure of the PE file. It contains global information about the program file and how to load it. Most of the information contained in this structure were more relivant to DOS software, and are only supported for backward compatability.

The structure follows the format:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    uint16_t e_magic;               // must contain "MZ"
    uint16_t e_cblp;          // number of bytes on the last page of the file
    uint16_t e_cp;            // number of pages in file
    uint16_t e_crlc;          // relocations
    uint16_t e_cparhdr;             // size of the header in paragraphs
    uint16_t e_minalloc;      // minimum and maximum paragraphs to allocate
    uint16_t e_maxalloc;
    uint16_t e_ss;            // initial SS:SP to set by Loader
    uint16_t e_sp;
    uint16_t e_csum;          // checksum
    uint16_t e_ip;            // initial CS:IP
    uint16_t e_cs;
    uint16_t e_lfarlc;        // address of relocation table
    uint16_t e_ovno;          // overlay number
    uint16_t e_res[4];        // resevered
    uint16_t e_oemid;               // OEM id
    uint16_t e_oeminfo;             // OEM info
    uint16_t e_res2[10];      // reserved
    uint32_t   e_lfanew;      // address of new EXE header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Alright, a lot of interesting things in this structure. The initial CS:IP and initial SS:SP members should be ignored as the operating system normally allocates a stack space and code descriptor value for CS. These members were more prominent during the DOS area and software requiring v8086 mode.

STUB Program

Okay then! Lets look back up at the PE file image structure again (The above picture.) Notice how a DOS stub program is right after the

IMAGE_DOS_HEADER structure. This is a useful program, actually. This is the program that displays "This program cannot be run in DOS Mode", if you try to execute a Windows program from within DOS.

We can change the stub program by using the **/STUB** linker option:

```
/stub=myprog.exe
```

When DOS attempts to load the executable, it will parse the IMAGE_DOS_HEADER structure and attempt to execute the DOS stub program because it is a valid DOS program. When running under the Win32 subsystem, the Windows loader will ignore the stub program.

IMAGE_NT_HEADERS

Following the STUB program is a structure, **IMAGE_NT_HEADERS** that contains the format of the PE header structures. Here is the structure:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

The signiture must match with "PE\0\0", where \0\0 are null characaters. The **IMAGE_FILE_HEADER** contains additional information used by the loader and the complete size of the **IMAGE_OPTIONAL_HEADER** structure. The **IMAGE_OPTIONAL_HEADER** is the largest and most important structure in the file. It also does not have a defined size.

In order to locate this structure, the OS loader must use the **e_lfanew** member of IMAGE_DOS_HEADER. **e_lfanew** is an RVA to this structure in memory, so in order to locate this structure, the loader needs to perform the following:

```
IMAGE_DOS_HEADER* pFile = (IMAGE_DOS_HEADER*) imageBase;
IMAGE_NT_HEADERS* pHeaders = (IMAGE_NT_HEADERS*) (pFile->e_lfanew + imageBase);
```

This assumes **imageBase** referres to the location where the program file was loaded into memory. Because older operating systems, such as DOS are not aware of this member of the header, it will be ignored by these OSs.

This structure contains the format for the other two header structures. Lets look at the first of these structures.

IMAGE_FILE_HEADER

The **IMAGE_FILE_HEADER** is the **Common Object File Format (COFF)** header structure. It follows the following format:

```
typedef struct _IMAGE_FILE_HEADER {
   USHORT  Machine;
   USHORT  NumberOfSections;                  // Number of sections in section table
   ULONG   TimeDateStamp;               // Date and time of program link
   ULONG   PointerToSymbolTable;        // RVA of symbol table
   ULONG   NumberOfSymbols;                   // Number of symbols in table
   USHORT  SizeOfOptionalHeader;        // Size of IMAGE_OPTIONAL_HEADER in bytes
   USHORT  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

This structure isnt too complex. Most of the above is only useful for debuggers (symbol table parsing). **SizeOfOptionalHeader** is important - because IMAGE_OPTIONAL_HEADER does not have a defined size, this member lets you know the size of the structure.

Machine can be one of the following values:

- 0x014c for x86 machines
- 0x0200 for x64 machines
- 0x8664 for AMD64 machines

In the usual case, it should be **0x014c** as we are developing for the x86 architecture.

Characteristics is composed of bit flags that can be bitwise-ORd by the linker to let the loader know different properties of the type of executable image this is. Heres the format:

- **Bit 0**: If set, image has no relocation information
- **Bit 1**: If set, File is executable
- **Bit 2**: If set, image has no COFF line numbers
- **Bit 3**: If set, image has no COFF symbol table entries
- **Bit 4**: If set, trim the working set for image. (Windows memory management specific. Obsolete)
- **Bit 5**: If set, loader assumes executable can handle >2GB VAs
- **Bit 6**: If set, loader assumes image supports 32 bit words
- **Bit 7**: If set, image has no debug information
- **Bit 8**: If set, image wont run directly from network drive (Windows specific)
- **Bit 9**: If set, image is treated as a SYSTEM file
- **Bit 10**: If set, image is treated as a DLL file
- **Bit 11**: If set, image will only run on single-processor machines
- **Bit 12**: If set, big-endian. obsolete flag

The Windows headers use defined constants, such as IMAGE_FILE_RELOCS_STRIPPED and IMAGE_FILE_EXECUTABLE_IMAGE that can be used when setting these flags.

As you can see, most of this structure is for information only for the loader on how to load the image. But wait! What about resources, symbol tables, debug info ... where is this at? Ah, behold the reason why **IMAGE_OPTIONAL_HEADER** does not have a defined size. Lets take a look!

IMAGE_OPTIONAL_HEADER

Ugh, here we go. This is the most complex structure in the file. The good news is that you probably have seen this structure before:

```
struct _IMAGE_OPTIONAL_HEADER {

   USHORT  Magic;                       // not-so-magical number
   UCHAR   MajorLinkerVersion;                    // linker version
   UCHAR   MinorLinkerVersion;
   ULONG   SizeOfCode;                  // size of .text in bytes
   ULONG   SizeOfInitializedData;       // size of .bss (and others) in bytes
   ULONG   SizeOfUninitializedData;               // size of .data,.sdata etc in bytes
   ULONG   AddressOfEntryPoint;         // RVA of entry point
   ULONG   BaseOfCode;                  // base of .text
   ULONG   BaseOfData;                  // base of .data
   ULONG   ImageBase;                   // image base VA
   ULONG   SectionAlignment;            // file section alignment
   ULONG   FileAlignment;               // file alignment
   USHORT  MajorOperatingSystemVersion;   // Windows specific. OS version required to run image
   USHORT  MinorOperatingSystemVersion;
   USHORT  MajorImageVersion;                     // version of program
   USHORT  MinorImageVersion;
   USHORT  MajorSubsystemVersion;                 // Windows specific. Version of SubSystem
   USHORT  MinorSubsystemVersion;
   ULONG   Reserved1;
   ULONG   SizeOfImage;                 // size of image in bytes
   ULONG   SizeOfHeaders;                   // size of headers (and stub program) in bytes
   ULONG   CheckSum;                    // checksum
   USHORT  Subsystem;                   // Windows specific. subsystem type
   USHORT  DllCharacteristics;                    // DLL properties
   ULONG   SizeOfStackReserve;                    // size of stack, in bytes
   ULONG   SizeOfStackCommit;                     // size of stack to commit
   ULONG   SizeOfHeapReserve;                     // size of heap, in bytes
   ULONG   SizeOfHeapCommit;                      // size of heap to commit
   ULONG   LoaderFlags;                 // no longer used
   ULONG   NumberOfRvaAndSizes;         // number of DataDirectory entries
   IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

First, take a look at that last member, **DataDirectory**. The constant, **IMAGE_NUMBEROF_DIRECTORY_ENTRIES** can, and has, changed through the years. This is that member that could change the size of this structure. We will look closer at that member a little later though as thats where all the fun stuff is at.

You might be interested in why this is called an "optional" header even though its clearly not optional. This is due to it being optional for COFF object files. While its not optional for executable images, it is for object files :)

magic can be one of the following:

- 0x10b: 32bit executable image
- 0x20b: 64bit executable image
- 0x107: ROM image

In the usual case, it should be **0x10b**.

A lot of the members in this structure really arent that complex.

The **subsystem** member is Windows-specific. It tells Windows what subsystem the program requires in order to execute properly. It can be one of the following values (posted here for completness only)

- 0: Unknown
- 1: Native SubSystem
- 2: GUI SubSystem
- 3: CUI SubSystem
- 5: OS/2 CUI SubSystem
- 7: POSIX CUI SubSystem
- 9: Windows CE GUI SubSystem
- 10: EFI
- 11: EFI Boot Driver
- 12: EFI Runtime Driver
- 13: EFI ROM
- 14: XBox
- 16: Boot application

The **DllCharacteristics** member contains bit flags that gives the loader information about the DLL. It follows the following format:

- **Bit 0-3**: reserved
- **Bit 4**: If set, DLL is relocatable
- **Bit 5**: If set, force code integrity checks
- **Bit 6**: If set, image is **Data Execution Prevention (DEP)** compatable
- **Bit 7**: If set, image should not be isolated
- **Bit 8**: If set, image does not use **Structured Exception Handling (SEH)**
- **Bit 9**: If set, image wont be binded
- **Bit 10**: reserved
- **Bit 11**: If set, image is a **Windows Driver Model (WDM)** driver
- **Bit 12**: reserved
- **Bit 13**: image is terminal server aware

AddressOfEntryPoint is an important one. This member contains the RVA of the entry point function of the image (for DLLs this can be null as DLLs dont need entry points.) This is what our bootloader uses to call our entry point function in our kernel.

Thats about all there is to it. You might be interested in what those other members are - for **.text**, **.data**, **.bss** etc. There is also that nasty looking **DataDirectory** member that we havnt looked at.

We will look at those members closly later on. For now, lets look at executing a program!

Executing a program

At this stage, **if all that you would like to do is execute a program**, all of the information has been provided. After loading a program, all that the loader needs to do is locate the **AddressOfEntryPoint** member from the optional header, and call that address. Remember that this is an RVA, meaning the loader needs to add this address to the **ImageBase** to obtain the linear address to the entry point function.

Here is an example:

```
//! loadedProgram is where the image was loaded to
IMAGE_DOS_HEADER* pImage = (IMAGE_DOS_HEADER*) loadedProgram;

//! go to NT HEADERS
IMAGE_NT_HEADERS* pHeaders = (IMAGE_NT_HEADERS*)(loadedProgram + pImage->e_lfanew);

//! get image base and entry point address from optional header
int base = pHeaders->OptionalHeader.ImageBase;
int entryPoint = pHeaders->OptionalHeader.AddressOfEntryPoint;

//! entry point function is at base+entryPoint
void (*entryFunction) () = (entryPoint + base);

//! call program entry point
entryFunction();
```

Thats all that is needed to execute a PE executable :)

Data Directories

Abstract

Resources, symbol tables, debugging information, import, export tables etc are accessable from that nifty **DataDirectory** member of the optional header. This member is an array of **IMAGE_DATA_DIRECTORY**'s that can be used to access other structures containing this information. **IMAGE_DATA_DIRECTORY** has the format:

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;        // RVA of table
```

```
  DWORD Size;                 // size of table
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Remember that **DataDirectory** is an **array** of **IMAGE_DATA_DIRECTORY**'s. Each entry in this array allows us to access the different data that we want to access.

Here are the index entries:

- 0: Export directory
- 1: Import directory
- 2: Resource directory
- 3: Exception directory
- 4: Security directory
- 5: Base relocation table
- 6: Debug directory
- 7: Description string
- 8: Machine value (MIPS GP)
- 9: TLS directory
- 10: Load configuation directory
- 14: COM+ data directory

For example, if you need to read the export table, reference **DataDirectory[0]**. If you want to read a resource, reference **DataDirectory[2].VirtualAddress**:

Each of these sections contains their own structures that are required to parse the specific data. Lets take a look at some of the more useful ones.

Reading the export table

The export table contains all functions exported from libraries or DLLs, including their function addresses within that DLL, their names, and ordinal number. The Win32 API function **GetProcAddress()** works by parsing the modules export table by ordinal number or name and returning the address from it. This is one way reading the export table can be useful.

To parse the export table, you need to first get the export directory structure. This is done by getting **DataDirectory[0]**.

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [0];
PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase);
```

Remember that **VirtualAddress** in the **IMAGE_DATA_DIRECTORY** structure is an RVA, so must be added to the image base. Now **exportDirectory** points to this nice structure:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  uint32_t Characteristics;
```

```
 uint32_t TimeDateStamp;
 uint16_t MajorVersion;
 uint16_t MinorVersion;
 uint32_t Name;
 uint32_t Base;
 uint32_t NumberOfFunctions;
 uint32_t NumberOfNames;
 uint32_t** AddressOfFunctions;
 uint32_t** AddressOfNames;
 uint16_t** AddressOfNameOrdinal;
}IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

This one is an easy one. **AddressOfFunctions** is an RVA that points to an array of function addresses. The function addresses, however are also RVAs. **AddressOfNames** is a pointer to a list of function names. All of these addresses are RVAs however so must be added to the image base in order to properly obtain the function name and address.

AddressOfNameOrdinal is an RVA to a list of ordinals. The ordinals, being just numbers representing the exported functions and not addresses, arent RVAs.

To properly parse the export table must be done in a loop. For example:

```
PDWORD FunctionNameAddressArray = ((DWORD)ExportDirectory->AddressOfNames) + ((PBYTE)imageBase);
PWORD  FunctionOrdinalAddressArray = (DWORD)ExportDirectory->AddressOfNameOrdinal + (PBYTE)imageBase;
PDWORD FunctionAddressArray = (DWORD)ExportDirectory->AddressOfFunctions + (PBYTE)imageBase;

//! search for function in exports table
for ( i = 0; i < ExportDirectory->NumberOfFunctions; i++ )
{
	LPSTR	FunctionName = FunctionNameAddressArray [i] + (PBYTE)imageBase;

	if (strcmp (FunctionName, funct) == 0) {

		WORD Ordinal = FunctionOrdinalAddressArray [i];
		DWORD FunctionAddress = FunctionAddressArray [Ordinal];
		return (PBYTE) (FunctionAddress + (PBYTE)imageBase);
	}
}
```

This can be used to impliment **GetProcAddress()** which can be useful in supporting DLLs.

Reading the import table

So... reading the export table wasnt hard enough, huh? Reading the import table isnt too hard, but is a little more involved then the export table. Ok, ok, whats the use for reading the import table? Its not so much the reading, but the **writing**. By writing entries into a programs inport table, you can allow function calls across libraries and DLLs without the need of a GetProcAddress() call. Windows performs this with delayed loaded DLLs and system DLLs.

In order to read the import table, you need to locate the import directory structure. This is at **DataDirectory[1]**:

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [1];
PIMAGE_IMPORT_DESCRIPTOR importDirectory = (PIMAGE_IMPORT_DESCRIPTOR) (DataDirectory->VirtualAddress + ImageBase);
```

It is important to note that **importDirectory** points to an **array** of descriptor entries. Each of these entries represents a module that was imported, such as an import DLL. Lets take a look at this structure:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
   union {
      uint32_t   Characteristics;            // 0 for terminating null import descriptor
      uint32_t   OriginalFirstThunk;         // RVA to INT
   };
   uint32_t   TimeDateStamp;                 // Time/Date of module, or other properties (see below)
   uint32_t   ForwarderChain;                // Forwarder chain ID
   uint32_t   Name;                  // Module name
   uint32_t   FirstThunk;            // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR *PIMAGE_IMPORT_DESCRIPTOR;
```

Its important to note that **Name, OriginalFirstThunk** and **FirstThunk** are RVAs. This means you will need to add the addresses (these are pointers) to the image base in order to properly parse the data. **Name** is an RVA that points to the imported module name, such as **kernel32.dll**. It is null terminated.

Remember that we are working with an array of import descriptors? How can we tell how many import descriptors that is in this array? The array ends with a null IMAGE_IMPORT_DESCRIPTOR, so an easy way to loop through each entry is this:

```
IMAGE_IMPORT_DESCRIPTOR* lpImportDesc;
while (! lpImportDesc->FirstThunk) {

      //! work with lpImportDesc here

      lpImportDesc++; // move to next entry
}
```

TimeDateStamp can be either the proper time/date or one of the following values:

- 0: module not bound
- -1: image is bound. Real time/date stamp stored

ForwarderChain is only used when supporting **DLL Forward Referencing**, which allows calls across DLLs to be forwarded to other DLLs. For example, some calls in Windows **kernel32.dll** are forwarded to other DLLs.

FirstThunk points to the IAT, **OriginalFirstThunk** points to an array of structures representing all imported functions. This is the **Import Name Table (INT)**. Both of these members are RVAs.

Thunk? right, Im sure you know another structure is coming up. Lets take a look:

```
typedef struct _IMAGE_THUNK_DATA {
	union {
		uint32_t* Function;			// address of imported function
		uint32_t  Ordinal;			// ordinal value of function
		PIMAGE_IMPORT_BY_NAME AddressOfData;		// RVA of imported name
		DWORD ForwarderStringl			// RVA to forwarder string
	} u1;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

OriginalFirstThunk are RVAs that point to an array of **IMAGE_THUNK_DATA** structures.

Ugh, yey, another structure. This one is a small one though:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
	uint16_t  Hint;		// Possible ordinal number to use
	uint8_t   Name[1];	// Name of function, null terminated
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Thats all there is to it. The first paramater can be 0, but it just hints the loader what ordinal number the function might be using. **Name** is an array of characters representing the name of the function.

Heres the deal: The IAT is just a list of addresses representing functions. What functions? The functions within this IMAGE_THUNK_DATA array. Look back at that IMAGE_THUNK_DATA structure and notice that its just an union representing a function name. This is the **Import Name Table (INT)**.

For example, lets say we want to get the current address of the function thats in IMAGE_THUNK_DATA[3]. Its address will be the 3rd dword in the IAT, which can be read using **IMAGE_IMPORT_DESCRIPTOR->FirstThunk**.

So, lets try to obtain the function name and address:

```
unsigned int count=0;
while (lpThunk->u1.Function) {

    //! get the function name
    char* lpFunctionName = (char*)((uint8_t*)imageBase + (uint32_t)lpThunk->u1.AddressOfData.Name);

    //! go into the IAT to get this functions address
    uint32_t* addr = (uint32_t*)((uint8_t*)imageBase + lpImportDesc->FirstThunk) + count;

    // lpFunctionName now points to the null terminated function name
    // addr now points to the address of this function

    count++;
    lpThunk++;
}
```

Image binding

This is where things get interesting. The IAT can be filled with the addresses of the imported functions either during **runtime** or **building** time. A **bounded image** is an image that has its IAT bounded to the functions during build time. An **unbounded** image is an image whose IAT is filled in by the OS loader during loading time.

If the image is bounded, you can do the following to call a function in an external DLL:

```

__declspec (dllimport) void function ();
function (); // calls myDll:function()
```

If the image is not bounded, the IAT contains junk. **It is then the responsbility of the OS loader to update the IAT** in order for the above code to work. This can be performed by reading the export table of the loaded DLL module (calling GetProcAddress(), and overwriting the IAT entry of that import function. Overwriting the IAT can be done by following the above - when you get the functions IAT entry, just overwrite it :)

This method can also be useful for installing **hooks** in DLLs and other modules.

Supporting resources

Introduction

Have you ever wondered how the Windows kernel can display an image and work with an XML configuation file without loading anything from disk? Have you ever worked with adding resources but wondered if it was possible to support them in an OS? The answer is a "yes, of course!"

Parsing resources is a bit more complex then the other directory types, however. Like the other sections, there is a base **IMAGE_RESOURCE_DIRECTORY** structure that can be obtained from the **DataDirectory** member of the optional header:

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [2];
PIMAGE_RESOURCE_DIRECTORY resourceDirectory = (PIMAGE_RESOURCE_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase);
```

Notice the pattern with how to access these sections? Oh, right, onto the new structure:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
   uint32_t              Characteristics;
   uint32_t              TimeDateStamp;
   uint16_t              MajorVersion;
   uint16_t              MinorVersion;
   uint16_t              NumberOfNamedEntries;
   uint16_t              NumberOfIdEntries;
   IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[1];
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

This structure doesnt have much of any interesting fields, except the last three.

If you have worked with Win32 resources, you might know that resources can be idenitified by ID or name. Two of the members in this structure will let us know the number of these entries, and the total amount of entries (NumberOfNamedEntries + NumberOfIdEntries), which is useful in looping through all of the entries. As you can probably guess, the entries are in the DirectoryEntries array. **DirectoryEntries** consists of an array of **IMAGE_RESOURCE_DIRECTORY_ENTRY** structures, which follow the format:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
   union {
      struct {
         DWORD NameOffset:31;
         DWORD NameIsString:1;
      };
      DWORD   Name;
      WORD    Id;
   };
   union {
      DWORD   OffsetToData;
      struct {
         DWORD   OffsetToDirectory:31;
         DWORD   DataIsDirectory:1;
      };
   };
} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Alright, this is an ugly structure. This structure represents a single resource, or resource directory.

Resource directory structure

resource or resource directory? Lets stop for a moment. (*grabs a cup of coffee*) ok, it is important to know that resources are stored as a **tree**. This tree is structured like this:

- Root directory
  - Resource group 1 Directory
    - Resource 1
    - Resource 2
  - Resource group 2 Directory
    - Resource 1
    - Resource 2
  - Resource group 3 Directory
    - Resource 1
    - Resource 2
  - ...etc...

There are a number of different resource groups which let us know the type of resources are in this group. Here are the group IDs:

- 1 - Cursor
- 2 - Bitmap
- 3 - Icon
- 4 - Menu
- 5 - Dialog
- 6 - String
- 7 - Font directory
- 8 - Font
- 9 - Accelerator
- 10 - RcData
- 11 - Message table
- 16 - Version
- 17 - DlgInclude/li>
- 19 - Plug and Play
- 20 - VXD
- 21 - Animated Cursor
- 22 - Animated Icon
- 23 - HTML
- 24 - Manifest

In order to locate a resource, you will need to traverse this tree. The good news is that this isnt hard if you assume there is only 3 layers in the tree.

First, lets look at looping through all of the entries in a resource directory:

```
//! get first entry in directory
IMAGE_RESOURCE_DIRECTORY_ENTRY* lpResourceEntry = lpResourceDir->DirectoryEntries;

//! loop through all entries
int entries = lpResourceDir->NumberOfIdEntries + lpResourceDir->NumberOfNamedEntries;
while (entries-- != 0) {

	//! look for bitmap resource (id=2)
	if (lpResourceEntry->Id == 2) {
		//! see below
	}
	lpResourceEntry++;
}
```

This is simple enough, huh? The **Id** member of **IMAGE_RESOURCE_DIRECTORY_ENTRY** is used to store the group ID. If we were looking for a bitmap, it would be in the bitmap group of the root directory, so look for the entry with ID=2.

Because IMAGE_RESOURCE_DIRECTORY_ENTRY represents both resource entries and directories, how can we tell what it is? Why, the **DataIsDirectory** member of course: If this member is set, its a directory. Ah, but if its a directory, how can you read the directory? Lets take a look:

```
if (lpResourceEntry->DataIsDirectory) {
	lpResourceEntry = lpResourceEntry->OffsetToDirectory;
	lpResourceEntry += startOfResourceSection;
}
```

This one isnt to bad. If the entry is a directory, the above obtains the offset to the new directory from the **OffsetToDirectory** and adds it to .. what? the *startOfResourceSection*!? Thats right... this is an offset, but not an RVA. I know ... *Why Microsoft, Why!?*

The start of the resource section is actually the address of the first member of the IMAGE_RESOURCE_DIRECTORY_ENTRY array. So by adding this address to the offset obtained from **OffsetToDirectory** you can obtain the pointer to the **IMAGE_RESOURCE_DIRECTORY** structure for this directory. Yes, then the whole process of reading those directory entries begins :)

If you are in the process of parsing the directory for your specific resource, just loop through all of the resource entries in the directory. If the resourceEntry ID field matches that of the resource ID you are trying to find (program specific ID here), then you have found the resource data.

The resource data is stored in a ... zomg! structure! It can be obtained from the **OffsetToData** member of the directory entry structure. Simular to the **OffsetToDirectory** member, this too is an offset from the start of the resource section.

Once you obtained the pointer, you can extract the resource data. Lets take a look at that structure:

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
```

```
    uint32_t   OffsetToData;
    uint32_t   Size;
    uint32_t   CodePage;
    uint32_t   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

Thats it! **OffsetToData** is an RVA to the real resource data, and **Size** is the size of that data, in bytes. For example, if we were looking for a bitmap resource, **OffsetToData** would be the RVA pointing to the bitmaps **BITMAPINFOHEADER** structure, which can be handled by any bitmap loader.

Conclusion

Thats all for this chapter. There are some planned updates to including covering additional sections (debug data and COMDATS) as well.

There is no demo for this chapter - it is primarily released for anyone that is interested in the internal workings of the PE executable file format and working with it. For the main series, we might only be loading and executing the program, so all of the other information is provided for completeness only. All code provided in text for demenstration has been tested (slightly modified) to work.

In the upcoming chapters, we will be using the PE executable file format and building a loader for supporting user mode programs. After that, on to multitasking!

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Chapter 23 Home