



DelphiBasics

Search this site[Home](#)[Counterstrikewi's Releases](#)[Snippets](#)[Projects](#)[Project Releases](#)[Articles](#)[Sitemap](#)

[Delphi Basics - Free Delphi Source Code - Ultimate Programming Resource](#) >
[Delphi Basics Articles](#) >

An In-Depth Look into the Win32 Portable Executable File Format - Part 2

posted 16 Mar 2010, 05:04 by Delphi Basics [updated 21 Nov 2010, 07:34]

See Part 1 here: [An In-Depth Look into the Win32 Portable Executable File Format - Part 1](#)

Source: <http://msdn.microsoft.com/en-us/magazine/cc301808.aspx>

SUMMARY The Win32 Portable Executable File Format (PE) was designed to be a standard executable format for use on all versions of the operating systems on all supported processors. Since its introduction, the PE format has undergone incremental changes, and the introduction of 64-bit Windows has required a few more. Part 1 of this series presented an overview and covered RVAs, the data directory, and the headers. This month in Part 2 the various sections of the executable are explored. The discussion includes the exports section, export forwarding, binding, and delayloading. The debug directory, thread local storage, and the resources sections are also covered.

Last month in [Part 1](#) of this article, I began a comprehensive tour of Portable Executable (PE) files. I described the history of PE files and the data structures that make up the headers, including the section table. The PE headers and section table tell you what kind of code and data exists in the executable and where you should look to find it.

This month I'll describe the more commonly encountered sections. If you're not familiar with basic PE file concepts, you should read Part 1 of this article first.

Last month I described how a section is a chunk of code or data that logically belongs together. For example, all the data that comprises an executable's import tables are in a section. Let's look at some of the sections you'll encounter in

executables and OBJs. Unless otherwise stated, the section names in **Figure 1** come from Microsoft tools.

Figure 1 Section Names

Name	Description
.text	The default code section.
.data	The default read/write data section. Global variables typically go here.
.rdata	The default read-only data section. String literals and C++/COM vtables are examples of items put into .rdata.
.idata	The imports table. It has become common practice (either explicitly, or via linker default behavior) to merge the .idata section into another section, typically .rdata. By default, the linker only merges the .idata section into another section when creating a release mode executable.
.edata	The exports table. When creating an executable that exports APIs or data, the linker creates an .EXP file. The .EXP file contains an .edata section that's added into the final executable. Like the .idata section, the .edata section is often found merged into the .text or .rdata sections.
.rsrc	The resources. This section is read-only. However, it should not be named anything other than .rsrc, and should not be merged into other sections.
.bss	Uninitialized data. Rarely found in executables created with recent linkers. Instead, the VirtualSize of the executable's .data section is expanded to make enough room for uninitialized data.
.crt	Data added for supporting the C++ runtime (CRT). A good example is the function pointers that are used to call the constructors and destructors of static C++ objects.
.tls	Data for supporting thread local storage variables declared with <code>__declspec(thread)</code> . This includes the initial value of the data, as well as additional variables needed by the runtime.
.reloc	The base relocations in an executable. Base relocations are generally only needed for DLLs and not EXEs. In release mode, the linker doesn't emit base relocations for EXE files. Relocations can be removed when linking with the /FIXED switch.

.sdata	"Short" read/write data that can be addressed relative to the global pointer. Used for the IA-64 and other architectures that use a global pointer register. Regular-sized global variables on the IA-64 will go in this section.
.srdata	"Short" read-only data that can be addressed relative to the global pointer. Used on the IA-64 and other architectures that use a global pointer register.
.pdata	The exception table. Contains an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures, which are CPU-specific. Pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION slot in the DataDirectory. Used for architectures with table-based exception handling, such as the IA-64. The only architecture that doesn't use table-based exception handling is the x86.
.debug\$S	Codeview format symbols in the OBJ file. This is a stream of variable-length CodeView format symbol records.
.debug\$T	Codeview format type records in the OBJ file. This is a stream of variable-length CodeView format type records.
.debug\$P	Found in the OBJ file when using precompiled headers.
.directve	Contains linker directives and is only found in OBJs. Directives are ASCII strings that could be passed on the linker command line. For instance: <code>-defaultlib:LIBC</code> Directives are separated by a space character.
.didat	Delayload import data. Found in executables built in nonrelease mode. In release mode, the delayload data is merged into another section.

The Exports Section

When an EXE exports code or data, it's making functions or variables usable by other EXEs. To keep things simple, I'll refer to exported functions and exported variables by the term "symbols." At a minimum, to export something, the address of an exported symbol needs to be obtainable in a defined manner. Each exported symbol has an ordinal number associated with it that can be used to look it up. Also, there is almost always an ASCII name associated with the symbol. Traditionally, the exported symbol name is the same as the name of the function or variable in the originating source file, although they can also be made to differ.

Typically, when an executable imports a symbol, it uses the symbol name rather than its ordinal. However, when importing by name, the system just uses the name to look up the export ordinal of the desired symbol, and retrieves the address using the ordinal value. It would be slightly faster if an ordinal had been used in the first place. Exporting and importing by name is solely a convenience for programmers.

The use of the `ORDINAL` keyword in the Exports section of a `.DEF` file tells the linker to create an import library that forces an API to be imported by ordinal, not by name.

I'll begin with the `IMAGE_EXPORT_DIRECTORY` structure, which is shown in **Figure 2**.

Figure 2 `IMAGE_EXPORT_DIRECTORY` Structure Members

Size	Member	Description
DWORD	Characteristics	Flags for the exports. Currently, none are defined.
DWORD	TimeDateStamp	The time/date that the exports were created. This field has the same definition as the <code>IMAGE_NT_HEADERS.FileHeader.TimeDateStamp</code> (number of seconds since 1/1/1970 GMT).
WORD	MajorVersion	The major version number of the exports. Not used, and set to 0.
WORD	MinorVersion	The minor version number of the exports. Not used, and set to 0.
DWORD	Name	A relative virtual address (RVA) to an ASCII string with the DLL name associated with these exports (for example, <code>KERNEL32.DLL</code>).
DWORD	Base	This field contains the starting ordinal value to be used for this executable's exports. Normally, this value is 1, but it's not required to be so. When looking up an export by ordinal, the value of this field is subtracted from the ordinal, with the result used as a zero-based index into the Export Address Table (EAT).

DWORD	NumberOfFunctions	The number of entries in the EAT. Note that some entries may be 0, indicating that no code/data is exported with that ordinal value.
DWORD	NumberOfNames	The number of entries in the Export Names Table (ENT). This value will always be less than or equal to the NumberOfFunctions field. It will be less when there are symbols exported by ordinal only. It can also be less if there are numeric gaps in the assigned ordinals. This field is also the size of the export ordinal table (below).
DWORD	AddressOfFunctions	The RVA of the EAT. The EAT is an array of RVAs. Each nonzero RVA in the array corresponds to an exported symbol.
DWORD	AddressOfNames	The RVA of the ENT. The ENT is an array of RVAs to ASCII strings. Each ASCII string corresponds to a symbol exported by name. This table is sorted so that the ASCII strings are in order. This allows the loader to do a binary search when looking for an exported symbol. The sorting of the names is binary (like the C++ RTL strcmp function provides), rather than a locale-specific alphabetic ordering.
DWORD	AddressOfNameOrdinals	The RVA of the export ordinal table. This table is an array of WORDs. This table maps an array index from the ENT into the corresponding export address table entry.

The exports directory points to three arrays and a table of ASCII strings. The only required array is the Export Address Table (EAT), which is an array of function pointers that contain the address of an exported function. An export ordinal is

simply an index into this array (see **Figure 3**).

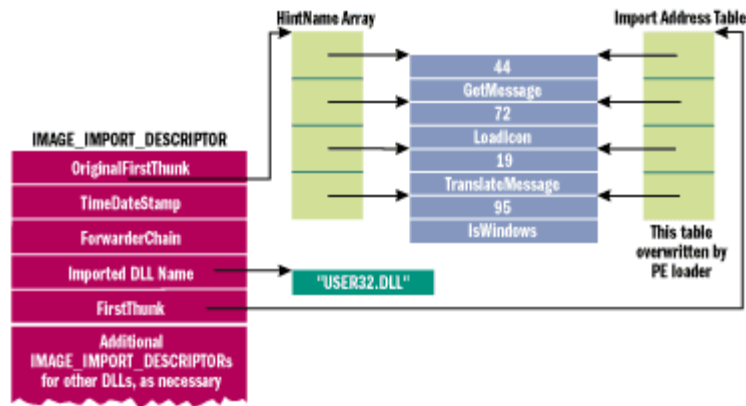


Figure 3 The IMAGE_EXPORT_DIRECTORY Structure

Let's go through an example to show exports at work. **Figure 4** shows some of the exports from `KERNEL32.DLL`.

Figure 4 KERNEL32 Exports

exports table:

```
Name:                KERNEL32.dll
Characteristics:      00000000
TimeDateStamp:       3B7DDFD8 -> Fri Aug 17 23:24:08 2001
Version:             0.00
Ordinal base:        00000001
# of functions:      000003A0
# of Names:          000003A0
```

```
Entry Pt  Ordnn  Name
00012ADA   1  ActivateActCtx
000082C2   2  AddAtomA
```

...remainder of exports omitted

Let's say you've called `GetProcAddress` on the `AddAtomA` API in `KERNEL32`. The system begins by locating `KERNEL32`'s `IMAGE_EXPORT_DIRECTORY`. From that, it obtains the start address of the Export Names Table (ENT). Knowing that there are `0x3A0` entries in the array, it does a binary search of the names until it finds the string `"AddAtomA"`.

Let's say that the loader finds `AddAtomA` to be the second array entry. The loader then reads the corresponding second value from the export ordinal table. This value is the export ordinal of `AddAtomA`. Using the export ordinal as an index into the EAT (and taking into account the Base field value), it turns out that `AddAtomA` is at a relative virtual address (RVA) of `0x82C2`. Adding `0x82C2` to the

load address of KERNEL32 yields the actual address of AddAtomA.

Export Forwarding

A particularly slick feature of exports is the ability to "forward" an export to another DLL. For example, in Windows NT®, Windows® 2000, and Windows XP, the KERNEL32 HeapAlloc function is forwarded to the RtlAllocHeap function exported by NTDLL. Forwarding is performed at link time by a special syntax in the EXPORTS section of the .DEF file. Using HeapAlloc as an example, KERNEL32's DEF file would contain:

```
EXPORTS
...
HeapAlloc = NTDLL.RtlAllocHeap
```

How can you tell if a function is forwarded rather than exported normally? It's somewhat tricky. Normally, the EAT contains the RVA of the exported symbol. However, if the function's RVA is inside the exports section (as given by the VirtualAddress and Size fields in the DataDirectory), the symbol is forwarded.

When a symbol is forwarded, its RVA obviously can't be a code or data address in the current module. Instead, the RVA points to an ASCII string of the DLL and symbol name to which it is forwarded. In the prior example, it would be NTDLL.RtlAllocHeap.

The Imports Section

The opposite of exporting a function or variable is importing it. In keeping with the prior section, I'll use the term "symbol" to collectively refer to imported functions and imported variables.

The anchor of the imports data is the IMAGE_IMPORT_DESCRIPTOR structure. The DataDirectory entry for imports points to an array of these structures. There's one IMAGE_IMPORT_DESCRIPTOR for each imported executable. The end of the IMAGE_IMPORT_DESCRIPTOR array is indicated by an entry with fields all set to 0. **Figure 5** shows the contents of an IMAGE_IMPORT_DESCRIPTOR.

Figure 5 IMAGE_IMPORT_DESCRIPTOR Structure

Size	Member	Description
DWORD	OriginalFirstThunk	This field is badly named. It contains the RVA of the Import Name Table (INT). This is an array of IMAGE_THUNK_DATA structures. This

		field is set to 0 to indicate the end of the array of IMAGE_IMPORT_DESCRIPTORs.
DWORD	TimeDateStamp	This is 0 if this executable is not bound against the imported DLL. When binding in the old style (see the section on Binding), this field contains the time/date stamp (number of seconds since 1/1/1970 GMT) when the binding occurred. When binding in the new style, this field is set to -1.
DWORD	ForwarderChain	This is the Index of the first forwarded API. Set to -1 if no forwarders. Only used for old-style binding, which could not handle forwarded APIs efficiently.
DWORD	Name	The RVA of the ASCII string with the name of the imported DLL.
DWORD	FirstThunk	Contains the RVA of the Import Address Table (IAT). This is array of IMAGE_THUNK_DATA structures.

Each IMAGE_IMPORT_DESCRIPTOR typically points to two essentially identical arrays. These arrays have been called by several names, but the two most common names are the Import Address Table (IAT) and the Import Name Table (INT).

Figure 6 shows an executable importing some APIs from USER32.DLL.

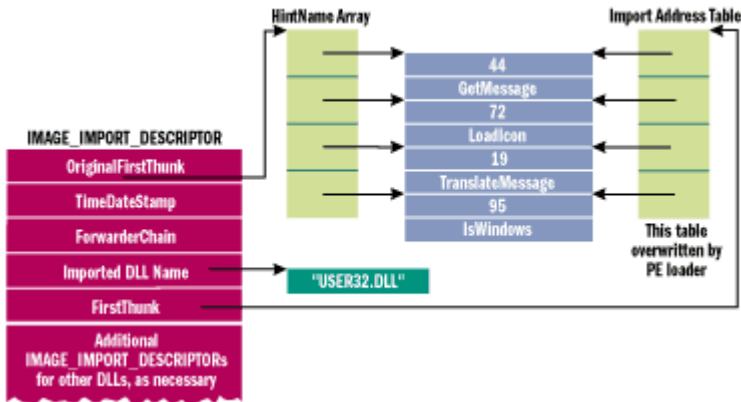


Figure 6 Two Parallel Arrays of Pointers

Both arrays have elements of type **IMAGE_THUNK_DATA**, which is a pointer-sized union. Each **IMAGE_THUNK_DATA** element corresponds to one imported function from the executable. The ends of both arrays are indicated by an

IMAGE_THUNK_DATA element with a value of zero. The IMAGE_THUNK_DATA union is a DWORD with these interpretations:

```
DWORD Function;      // Memory address of the imported function
DWORD Ordinal;       // Ordinal value of imported API
DWORD AddressOfData; // RVA to an IMAGE_IMPORT_BY_NAME with
                    // the imported API name
DWORD ForwarderString; // RVA to a forwarder string
```

The IMAGE_THUNK_DATA structures within the IAT lead a dual-purpose life. In the executable file, they contain either the ordinal of the imported API or an RVA to an IMAGE_IMPORT_BY_NAME structure. The IMAGE_IMPORT_BY_NAME structure is just a WORD, followed by a string naming the imported API. The WORD value is a "hint" to the loader as to what the ordinal of the imported API might be. When the loader brings in the executable, it overwrites each IAT entry with the actual address of the imported function. This is a key point to understand before proceeding. I highly recommend reading Russell Osterlund's article in this issue which describes the steps that the Windows loader takes.

Before the executable is loaded, is there a way you can tell if an IMAGE_THUNK_DATA structure contains an import ordinal, as opposed to an RVA to an IMAGE_IMPORT_BY_NAME structure? The key is the high bit of the IMAGE_THUNK_DATA value. If set, the bottom 31 bits (or 63 bits for a 64-bit executable) is treated as an ordinal value. If the high bit isn't set, the IMAGE_THUNK_DATA value is an RVA to the IMAGE_IMPORT_BY_NAME.

The other array, the INT, is essentially identical to the IAT. It's also an array of IMAGE_THUNK_DATA structures. The key difference is that the INT isn't overwritten by the loader when brought into memory. Why have two parallel arrays for each set of APIs imported from a DLL? The answer is in a concept called binding. When the binding process rewrites the IAT in the file (I'll describe this process later), some way of getting the original information needs to remain. The INT, which is a duplicate copy of the information, is just the ticket.

An INT isn't required for an executable to load. However, if not present, the executable cannot be bound. The Microsoft linker seems to always emit an INT, but for a long time, the Borland linker (TLINK) did not. The Borland-created files could not be bound.

In early Microsoft linkers, the imports section wasn't all that special to the linker. All the data that made up an executable's imports came from import libraries. You could see this for yourself by running Dumpbin or PEDUMP on an import library. You'd find sections with names like .idata\$3 and .idata\$4. The linker simply followed its rules for combining sections, and all the structures and arrays magically fell into place. A few years back, Microsoft introduced a new import library format that creates significantly smaller import libraries at the cost of the linker taking a more active role in creating the import data.

Binding

When an executable is bound (via the Bind program, for instance), the `IMAGE_THUNK_DATA` structures in the IAT are overwritten with the actual address of the imported function. The executable file on disk has the actual in-memory addresses of APIs in other DLLs in its IAT. When loading a bound executable, the Windows loader can bypass the step of looking up each imported API and writing it to the IAT. The correct address is already there! This only happens if the stars align properly, however. My May 2000 column contains some benchmarks on just how much load-time speed increase you can get from binding executables.

You probably have a healthy skepticism about the safety of executable binding. After all, what if you bind your executable and the DLLs that it imports change? When this happens, all the addresses in the IAT are invalid. The loader checks for this situation and reacts accordingly. If the addresses in the IAT are stale, the loader still has all the necessary information from the INT to resolve the addresses of the imported APIs.

Binding your programs at installation time is the best possible scenario. The `BindImage` action of the Windows installer will do this for you. Alternatively, `IMAGEHLP.DLL` provides the `BindImageEx` API. Either way, binding is good idea. If the loader determines that the binding information is current, executables load faster. If the binding information becomes stale, you're no worse off than if you hadn't bound in the first place.

One of the key steps in making binding effective is for the loader to determine if the binding information in the IAT is current. When an executable is bound, information about the referenced DLLs is placed into the executable. The loader checks this information to make a quick determination of the binding validity. This information wasn't added with the first implementation of binding. Thus, an executable can be bound in the old way or the new way. The new way is what I'll describe here.

The key data structure in determining the validity of bound imports is an `IMAGE_BOUND_IMPORT_DESCRIPTOR`. A bound executable contains a list of these structures. Each `IMAGE_BOUND_IMPORT_DESCRIPTOR` structure represents the time/date stamp of one imported DLL that has been bound against. The RVA of the list is given by the `IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT` element in the `DataDirectory`. The elements of the `IMAGE_BOUND_IMPORT_DESCRIPTOR` are:

- `TimeStamp`, a `DWORD` that contains the time/date stamp of the imported DLL.
- `OffsetModuleName`, a `WORD` that contains an offset to a string with the name of the imported DLL. This field is an offset (not an RVA) from the first `IMAGE_BOUND_IMPORT_DESCRIPTOR`.
- `NumberOfModuleForwarderRefs`, a `WORD` that contains the number of `IMAGE_BOUND_FORWARDER_REF` structures that immediately follow this

structure. These structures are identical to the `IMAGE_BOUND_IMPORT_DESCRIPTOR` except that the last WORD (the `NumberOfModuleForwarderRefs`) is reserved.

In a simple world, the `IMAGE_BOUND_IMPORT_DESCRIPTOR`s for each imported DLL would be a simple array. But, when binding against an API that's forwarded to another DLL, the validity of the forwarded DLL has to be checked too. Thus, the `IMAGE_BOUND_FORWARDER_REF` structures are interleaved with the `IMAGE_BOUND_IMPORT_DESCRIPTOR`s.

Let's say you linked against `HeapAlloc`, which is forwarded to `RtlAllocateHeap` in `NTDLL`. Then you ran `BIND` on your executable. In your EXE, you'd have an `IMAGE_BOUND_IMPORT_DESCRIPTOR` for `KERNEL32.DLL`, followed by an `IMAGE_BOUND_FORWARDER_REF` for `NTDLL.DLL`. Immediately following that might be additional `IMAGE_BOUND_IMPORT_DESCRIPTOR`s for other DLLs you imported and bound against.

Delayload Data

Earlier I described how delayloading a DLL is a hybrid approach between an implicit import and explicitly importing APIs via `LoadLibrary` and `GetProcAddress`. Now let's take a look at the data structures and see how delayloading works.

Remember that delayloading is not an operating system feature. It's implemented entirely by additional code and data added by the linker and runtime library. As such, you won't find many references to delayloading in `WINNT.H`. However, you can see definite parallels between the delayload data and regular imports data.

The delayload data is pointed to by the `IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT` entry in the `DataDirectory`. This is an RVA to an array of `ImgDelayDescr` structures, defined in `DelayImp.H` from Visual C++. Figure 7 shows the contents. There's one `ImgDelayDescr` for each delayload imported DLL.

Figure 7 `ImgDelayDescr` Structure

Size	Member	Description
DWORD	grAttrs	The attributes for this structure. Currently, the only flag defined is <code>dlattrRva</code> (1), indicating that the address fields in the structure should be treated as RVAs, rather than virtual addresses.
RVA	rvaDLLName	An RVA to a string with the name of the imported DLL. This string is passed to <code>LoadLibrary</code> .

RVA	rvaHmod	An RVA to an HMODULE-sized memory location. When the Delayloaded DLL is brought into memory, its HMODULE is stored at this location.
RVA	rvaIAT	An RVA to the Import Address Table for this DLL. This is the same format as a regular IAT.
RVA	rvaINT	An RVA to the Import Name Table for this DLL. This is the same format as a regular INT.
RVA	rvaBoundIAT	An RVA of the optional bound IAT. An RVA to a bound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently, this copy of the IAT is not actually bound, but this feature may be added in future versions of the BIND program.
RVA	rvaUnloadIAT	An RVA of the optional copy of the original IAT. An RVA to an unbound copy of an Import Address Table for this DLL. This is the same format as a regular IAT. Currently always set to 0.
DWORD	dwTimeStamp	The date/time stamp of the delayload imported DLL. Normally set to 0.

The key thing to glean from `ImgDelayDescr` is that it contains the addresses of an IAT and an INT for the DLL. These tables are identical in format to their regular imports equivalent, only they're written to and read by the runtime library code rather than the operating system. When you call an API from a delayloaded DLL for the first time, the runtime calls `LoadLibrary` (if necessary), and then `GetProcAddress`. The resulting address is stored in the delayload IAT so that future calls go directly to the API.

There is a bit of goofiness about the delayload data that needs explanation. In its original incarnation in Visual C++ 6.0, all `ImgDelayDescr` fields containing addresses used virtual addresses, rather than RVAs. That is, they contained actual addresses where the delayload data could be found. These fields are DWORDs, the size of a pointer on the x86.

Now fast-forward to IA-64 support. All of a sudden, 4 bytes isn't enough to hold a complete address. Oops! At this point, Microsoft did the correct thing and changed the fields containing addresses to RVAs. As shown in Figure 7, I've used the revised structure definitions and names.

There is still the issue of determining whether an `ImgDelayDescr` is using RVAs or virtual addresses. The structure has a field to hold flag values. When the "1" bit of the `grAttrs` field is on, the structure members should be treated as RVAs. This is the only option starting with Visual Studio® .NET and the 64-bit compiler. If that bit in `grAttrs` is off, the `ImgDelayDescr` fields are virtual addresses.

The Resources Section

Of all the sections within a PE, the resources are the most complicated to navigate. Here, I'll describe just the data structures that are used to get to the raw resource data such as icons, bitmaps, and dialogs. I won't go into the actual format of the resource data since it's beyond the scope of this article.

The resources are found in a section called `.rsrc`. The `IMAGE_DIRECTORY_ENTRY_RESOURCE` entry in the `DataDirectory` contains the RVA and size of the resources. For various reasons, the resources are organized in a manner similar to a file system—with directory and leaf nodes.

The resource pointer from the `DataDirectory` points to a structure of type `IMAGE_RESOURCE_DIRECTORY`. The `IMAGE_RESOURCE_DIRECTORY` structure contains unused `Characteristic`, `TimeStamp`, and version number fields. The only interesting fields in an `IMAGE_RESOURCE_DIRECTORY` are the `NumberOfNamedEntries` and the `NumberOfIdEntries`.

Following each `IMAGE_RESOURCE_DIRECTORY` structure is an array of `IMAGE_RESOURCE_DIRECTORY_ENTRY` structures. Adding the `NumberOfNamedEntries` and `NumberOfIdEntries` fields from the `IMAGE_RESOURCE_DIRECTORY` yields the count of `IMAGE_RESOURCE_DIRECTORY_ENTRY`s. (If all these data structure names are painful for you to read, let me tell you, it's also awkward writing about them!)

A directory entry points to either another resource directory or to the data for an individual resource. When the directory entry points to another resource directory, the high bit of the second `DWORD` in the structure is set and the remaining 31 bits are an offset to the resource directory. The offset is relative to the beginning of the resource section, not an RVA.

When a directory entry points to an actual resource instance, the high bit of the second `DWORD` is clear. The remaining 31 bits are the offset to the resource instance (for example, a dialog). Again, the offset is relative to the resource section, not an RVA.

Directory entries can be named or identified by an ID value. This is consistent with resources in an `.RC` file where you can specify a name or an ID for a resource instance. In the directory entry, when the high bit of the first `DWORD` is set, the remaining 31 bits are an offset to the string name of the resource. If the high bit is clear, the bottom 16 bits contain the ordinal identifier.

Enough theory! Let's look at an actual resource section and decipher what it

means. **Figure 8** shows abbreviated PEDUMP output for the resources in ADVAPI32.DLL.

Figure 8 Resources from ADVAPI32.DLL

Resources (RVA: 6B000)

ResDir (0) Entries:03 (Named:01, ID:02) TimeDate:00000000

 ResDir (MOFDATA) Entries:01 (Named:01, ID:00) TimeDate:00000000

 ResDir (MOFRESOURCE_NAME) Entries:01 (Named:00, ID:01) TimeDate:00

 ID: 00000409 DataEntryOffs: 00000128

 DataRVA: 6B6F0 DataSize: 190F5 CodePage: 0

 ResDir (STRING) Entries:01 (Named:00, ID:01) TimeDate:00000000

 ResDir (C36) Entries:01 (Named:00, ID:01) TimeDate:00000000

 ID: 00000409 DataEntryOffs: 00000138

 DataRVA: 6B1B0 DataSize: 0053C CodePage: 0

 ResDir (RCDATA) Entries:01 (Named:00, ID:01) TimeDate:00000000

 ResDir (66) Entries:01 (Named:00, ID:01) TimeDate:00000000

 ID: 00000409 DataEntryOffs: 00000148

 DataRVA: 85908 DataSize: 0005C CodePage: 0

Each line that starts with "ResDir" corresponds to an IMAGE_RESOURCE_DIRECTORY structure. Following "ResDir" is the name of the resource directory, in parentheses. In this example, there are resource directories named 0, MOFDATA, MOFRESOURCE_NAME, STRING, C36, RCDATA, and 66. Following the name is the combined number of directory entries (both named and by ID). In this example, the topmost directory has three immediate directory entries, while all the other directories contain a single entry.

In everyday use, the topmost directory is analogous to the root directory of a file system. Each directory entry below the "root" is always a directory in its own right. Each of these second-level directories corresponds to a resource type (strings tables, dialogs, menus, and so on). Underneath each of the second-level "resource type" directories, you'll find third-level subdirectories.

There's a third-level subdirectory for each resource instance. For example, if there were five dialogs, there would be a second-level DIALOG directory with five directory entries beneath it. Each of the five directory entries would themselves be a directory. The name of the directory entry corresponds to the name or ID of the resource instance. Under each of these directory entries is a single item which contains the offset to the resource data. Simple, no?

If you learn more efficiently by reading code, be sure to check out the resource dumping code in PEDUMP (see the February 2002 code download for this article). Besides displaying all the resource directories and their entries, it also dumps out

several of the more common types of resource instances such as dialogs.

Base Relocations

In many locations in an executable, you'll find memory addresses. When an executable is linked, it's given a preferred load address. These memory addresses are only correct if the executable loads at the preferred load address specified by the ImageBase field in the IMAGE_FILE_HEADER structure.

If the loader needs to load the DLL at another address, all the addresses in the executable will be incorrect. This entails extra work for the loader. The May 2000 Under The Hood column (mentioned earlier) describes the performance hit when DLLs have the same preferred load addresses and how the REBASE tool can help.

The base relocations tell the loader every location in the executable that needs to be modified if the executable doesn't load at the preferred load address. Luckily for the loader, it doesn't need to know any details about how the address is being used. It just knows that there's a list of locations that need to be modified in some consistent way.

Let's look at an x86-based example to make this clear. Say you have the following instruction, which loads the value of a local variable (at address 0x0040D434) into the ECX register:

```
00401020: 8B 0D 34 D4 40 00  mov ecx,dword ptr [0x0040D434]
```

The instruction is at address 0x00401020 and is six bytes long. The first two bytes (0x8B 0x0D) make up the opcode of the instruction. The remaining four bytes hold a DWORD address (0x0040D434). In this example, the instruction is from an executable with a preferred load address of 0x00400000. The global variable is therefore at an RVA of 0xD434.

If the executable does load at 0x00400000, the instruction can run exactly as is. But let's say that the executable somehow gets loaded at address of 0x00500000. If this happens, the last four bytes of the instruction need to be changed to 0x0050D434.

How can the loader make this change? The loader compares the preferred and actual load addresses and calculates a delta. In this case, the delta value is 0x00100000. This delta can be added to the value of the DWORD-sized address to come up with the new address of the variable. In the previous example, there would be a base relocation for address 0x00401022, which is the location of the DWORD in the instruction.

In a nutshell, base relocations are just a list of locations in an executable where a delta value needs to be added to the existing contents of memory. The pages of an executable are brought into memory only as they're needed, and the format of the base relocations reflects this. The base relocations reside in a section called .reloc, but the correct way to find them is from the DataDirectory using the IMAGE_DIRECTORY_ENTRY_BASERELOC entry.

Base relocations are a series of very simple `IMAGE_BASE_RELOCATION` structures. The `VirtualAddress` field contains the RVA of the memory range to which the relocations belong. The `SizeOfBlock` field indicates how many bytes make up the relocation information for this base, including the size of the `IMAGE_BASE_RELOCATION` structure.

Immediately following the `IMAGE_BASE_RELOCATION` structure is a variable number of WORD values. The number of WORDs can be deduced from the `SizeOfBlock` field. Each WORD consists of two parts. The top 4 bits indicate the type of relocation, as given by the `IMAGE_REL_BASED_XXX` values in `WINNT.H`. The bottom 12 bits are an offset, relative to the `VirtualAddress` field, where the relocation should be applied.

In the previous example of base relocations, I simplified things a bit. There are actually multiple types of base relocations and methods for how they're applied. For x86 executables, all base relocations are of type `IMAGE_REL_BASED_HIGHLOW`. You will often see a relocation of type `IMAGE_REL_BASED_ABSOLUTE` at the end of a group of relocations. These relocations do nothing, and are there just to pad things so that the next `IMAGE_BASE_RELOCATION` is aligned on a 4-byte boundary.

For IA-64 executables, the relocations seem to always be of type `IMAGE_REL_BASED_DIR64`. As with x86 relocations, there will often be `IMAGE_REL_BASED_ABSOLUTE` relocations used for padding. Interestingly, although pages in IA-64 EXEs are 8KB, the base relocations are still done in 4KB chunks.

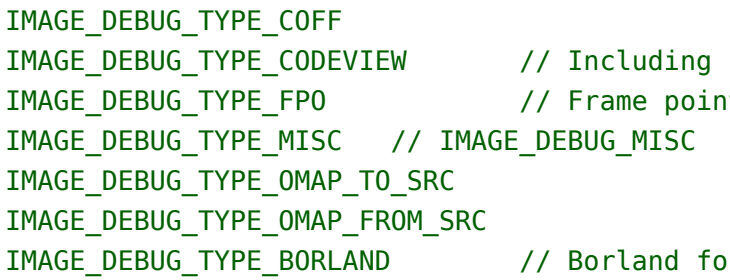
In Visual C++ 6.0, the linker omits relocations for EXEs when doing a release build. This is because EXEs are the first thing brought into an address space, and therefore are essentially guaranteed to load at the preferred load address. DLLs aren't so lucky, so base relocations should always be left in, unless you have a reason to omit them with the `/FIXED` switch. In Visual Studio .NET, the linker omits base relocations for debug and release mode EXE files.

The Debug Directory

When an executable is built with debug information, it's customary to include details about the format of the information and where it is. The operating system doesn't require this to run the executable, but it's useful for development tools. An EXE can have multiple forms of debug information; a data structure known as the debug directory indicates what's available.

The `DebugDirectory` is found via the `IMAGE_DIRECTORY_ENTRY_DEBUG` slot in the `DataDirectory`. It consists of an array of `IMAGE_DEBUG_DIRECTORY` structures (see **Figure 9**), one for each type of debug information. The number of elements in the debug directory can be calculated using the `Size` field in the `DataDirectory`.

Figure 9 Fields of IMAGE_DEBUG_DIRECTORY

Size	Member	Description
DWORD	Characteristics	Unused and set to 0.
DWORD	TimeStamp	The time/date stamp of this debug information (number since 1/1/1970, GMT).
WORD	MajorVersion	The major version of this debug information. Unused.
WORD	MinorVersion	The minor version of this debug information. Unused.
DWORD	Type	The type of the debug information. The following type most commonly encountered:  <pre> IMAGE_DEBUG_TYPE_COFF IMAGE_DEBUG_TYPE_CODEVIEW // Including IMAGE_DEBUG_TYPE_FPO // Frame poin IMAGE_DEBUG_TYPE_MISC // IMAGE_DEBUG_MISC IMAGE_DEBUG_TYPE_OMAP_TO_SRC IMAGE_DEBUG_TYPE_OMAP_FROM_SRC IMAGE_DEBUG_TYPE_BORLAND // Borland fo </pre>
DWORD	SizeOfData	The size of the debug data in this file. Doesn't count the external debug files such as .PDBs.
DWORD	AddressOfRawData	The RVA of the debug data, when mapped into memory; if the debug data isn't mapped in.
DWORD	PointerToRawData	The file offset of the debug data (not an RVA).

By far, the most prevalent form of debug information today is the PDB file. The PDB file is essentially an evolution of CodeView-style debug information. The presence of PDB information is indicated by a debug directory entry of type `IMAGE_DEBUG_TYPE_CODEVIEW`. If you examine the data pointed to by this entry, you'll find a short CodeView-style header. The majority of this debug data is just a path to the external PDB file. In Visual Studio 6.0, the debug header began with an NB10 signature. In Visual Studio .NET, the header begins with an RSIDS.

In Visual Studio 6.0, COFF debug information can be generated with the `/DEBUGTYPE:COFF` linker switch. This capability is gone in Visual Studio .NET. Frame Pointer Omission (FPO) debug information comes into play with optimized x86 code, where the function may not have a regular stack frame. FPO data allows the debugger to locate local variables and parameters.

The two types of OMAP debug information exist only for Microsoft programs. Microsoft has an internal tool that reorganizes the code in executable files to minimize paging. (Yes, more than the Working Set Tuner can do.) The OMAP information lets tools convert between the original addresses in the debug information and the new addresses after having been moved.

Incidentally, DBG files also contain a debug directory like I just described. DBG files were prevalent in the Windows NT 4.0 era, and they contained primarily COFF debug information. However, they've been phased out in favor of PDB files in Windows XP.

The .NET Header

Executables produced for the Microsoft .NET environment are first and foremost PE files. However, in most cases normal code and data in a .NET file are minimal. The primary purpose of a .NET executable is to get the .NET-specific information such as metadata and intermediate language (IL) into memory. In addition, a .NET executable links against MSCOREE.DLL. This DLL is the starting point for a .NET process. When a .NET executable loads, its entry point is usually a tiny stub of code. That stub just jumps to an exported function in MSCOREE.DLL (`_CorExeMain` or `_CorDllMain`). From there, MSCOREE takes charge, and starts using the metadata and IL from the executable file. This setup is similar to the way apps in Visual Basic (prior to .NET) used MSVBVM60.DLL. The starting point for .NET information is the `IMAGE_COR20_HEADER` structure, currently defined in `CorHdr.h` from the .NET Framework SDK and more recent versions of `WINNT.H`. The `IMAGE_COR20_HEADER` is pointed to by the `IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR` entry in the `DataDirectory`. **Figure 10** shows the fields of an `IMAGE_COR20_HEADER`. The format of the metadata, method IL, and other things pointed to by the `IMAGE_COR20_HEADER` will be described in a subsequent article.

Figure 10 `IMAGE_COR20_HEADER` Structure

Type	Member	Description
DWORD	cb	Size of the header in bytes.
WORD	MajorRuntimeVersion	The minimum version of the first release of .NET, this va
WORD	MinorRuntimeVersion	The minor portion of the ver
IMAGE_DATA_DIRECTORY	MetaData	The RVA to the metadata tal
DWORD	Flags	Flag values containing attri defined as: <code>COMIMAGE_FLAGS_ILONLY</code> // // <code>COMIMAGE_FLAGS_32BITREQ</code> <code>COMIMAGE_FLAGS_IL_LIBRARY</code> <code>STRONGNAMESIGNED</code> // <code>COMIMAGE_FLAGS_TRACKDEB</code>

DWORD	EntryPointToken	Token for the MethodDef of calls this method to begin n
IMAGE_DATA_DIRECTORY	Resources	The RVA and size of the .NE
IMAGE_DATA_DIRECTORY	StrongNameSignature	The RVA of the strong name
IMAGE_DATA_DIRECTORY	CodeManagerTable	The RVA of the code manag required to obtain the state and track GC references).
IMAGE_DATA_DIRECTORY	VTableFixups	The RVA of an array of func of unmanaged C++ vtables.
IMAGE_DATA_DIRECTORY	ExportAddressTableJumps	The RVA to an array of RVAs thunks allow managed meth call them.
IMAGE_DATA_DIRECTORY	ManagedNativeHeader	For internal use of the .NET

TLS Initialization

When using thread local variables declared with `__declspec(thread)`, the compiler puts them in a section named `.tls`. When the system sees a new thread starting, it allocates memory from the process heap to hold the thread local variables for the thread. This memory is initialized from the values in the `.tls` section. The system also puts a pointer to the allocated memory in the TLS array, pointed to by `FS:[2Ch]` (on the x86 architecture).

The presence of thread local storage (TLS) data in an executable is indicated by a nonzero `IMAGE_DIRECTORY_ENTRY_TLS` entry in the `DataDirectory`. If nonzero, the entry points to an `IMAGE_TLS_DIRECTORY` structure, shown in **Figure 11**.

Figure 11 IMAGE_TLS_DIRECTORY Structure

Size	Member	Description
DWORD	StartAddressOfRawData	The beginning address of a range of memory used to initialize a new thread's TLS data in memory.
DWORD	EndAddressOfRawData	The ending address of the range of memory used to initialize a new thread's TLS data in memory.

DWORD	AddressOfIndex	When the executable is brought into memory and a .tls section is present, the loader allocates a TLS handle via TlsAlloc. It stores the handle at the address given by this field. The runtime library uses this index to locate the thread local data.
DWORD	AddressOfCallBacks	Address of an array of PIMAGE_TLS_CALLBACK function pointers. When a thread is created or destroyed, each function in the list is called. The end of the list is indicated by a pointer-sized variable set to 0. In normal Visual C++ executables, this list is empty.
DWORD	SizeOfZeroFill	The size in bytes of the initialization data, beyond the initialized data delimited by the StartAddressOfRawData and EndAddressOfRawData fields. All per-thread data after this range is initialized to 0.
DWORD	Characteristics	Reserved. Currently set to 0.

It's important to note that the addresses in the IMAGE_TLS_DIRECTORY structure are virtual addresses, not RVAs. Thus, they will get modified by base relocations if the executable doesn't load at its preferred load address. Also, the IMAGE_TLS_DIRECTORY itself is not in the .tls section; it resides in the .rdata section.

Program Exception Data

Some architectures (including the IA-64) don't use frame-based exception handling, like the x86 does; instead, they used table-based exception handling in which there is a table containing information about every function that might be affected by exception unwinding. The data for each function includes the starting address, the ending address, and information about how and where the exception should be handled. When an exception occurs, the system searches through the tables to locate the appropriate entry and handles it. The exception table is an array of IMAGE_RUNTIME_FUNCTION_ENTRY structures. The array is pointed to by the

IMAGE_DIRECTORY_ENTRY_EXCEPTION entry in the DataDirectory. The format of the IMAGE_RUNTIME_FUNCTION_ENTRY structure varies from architecture to architecture. For the IA-64, the layout looks like this:

```
DWORD BeginAddress;  
DWORD EndAddress;  
DWORD UnwindInfoAddress;
```

The format of the UnwindInfoAddress data isn't given in WINNT.H. However, the format can be found in Chapter 11 of the "IA-64 Software Conventions and Runtime Architecture Guide" from Intel.

Wrap-up

The Portable Executable format is a well-structured and relatively simple executable format. It's particularly nice that PE files can be mapped directly into memory so that the data structures on disk are the same as those Windows uses at runtime. I've also been surprised at how well the PE format has held up with all the various changes that have been thrown at it in the past 10 years, including the transition to 64-bit Windows and .NET.

Although I've covered many aspects of PE files, there are still topics that I haven't gotten to. There are flags, attributes, and data structures that occur infrequently enough that I decided not to describe them here. However, I hope that this "big picture" introduction to PE files has made the Microsoft PE specifications easier for you to understand.

Comments

You do not have permission to add comments.

