Peering Inside the PE: A Tour of the Win32 Portable Executable File Format

Matt Pietrek

March 1994

Matt Pietrek is the author of Windows Internals *(Addison-Wesley, 1993). He works at Nu-Mega Technologies Inc., and can be reached via CompuServe: 71774,362*

The format of an operating system's executable file is in many ways a mirror of the operating system. Although studying an executable file format isn't usually high on most programmers' list of things to do, a great deal of knowledge can be gleaned this way. In this article, I'll give a tour of the Portable Executable (PE) file format that Microsoft has designed for use by all their Win32®-based systems: Windows NT®, Win32s™, and Windows® 95. The PE format plays a key role in all of Microsoft's operating systems for the foreseeable future, including Windows 2000. If you use Win32s or Windows NT, you're already using PE files. Even if you program only for Windows 3.1 using Visual C++®, you're still using PE files (the 32-bit MS-DOS® extended components of Visual C++ use this format). In short, PEs are already pervasive and will become unavoidable in the near future. Now is the time to find out what this new type of executable file brings to the operating system party.

I'm not going to make you stare at endless hex dumps and chew over the significance of individual bits for pages on end. Instead, I'll present the concepts embedded in the PE file format and relate them to things you encounter everyday. For example, the notion of thread local variables, as in

```
declspec(thread) int i;
```

drove me crazy until I saw how it was implemented with elegant simplicity in the executable file. Since many of you are coming from a background in 16-bit Windows, I'll correlate the constructs of the Win32 PE file format back to their 16-bit NE file format equivalents.

In addition to a different executable format, Microsoft also introduced a new object module format produced by their compilers and assemblers. This new OBJ file format has many things in common with the PE executable format. I've searched in vain to find any documentation on the new OBJ file format. So I deciphered it on my own, and will describe parts of it here in addition to the PE format.

It's common knowledge that Windows NT has a VAX® VMS® and UNIX® heritage. Many of the Windows NT creators designed and coded for those platforms before coming to Microsoft. When it came time to design Windows NT, it was only natural that they tried to minimize their bootstrap time by using previously written and tested tools. The executable and object module format that these tools produced and worked with is called COFF (an acronym for Common Object File Format). The relative age

of COFF can be seen by things such as fields specified in octal format. The COFF format by itself was a good starting point, but needed to be extended to meet all the needs of a modern operating system like Windows NT or Windows 95. The result of this updating is the Portable Executable format. It's called "portable" because all the implementations of Windows NT on various platforms (x86, MIPS®, Alpha, and so on) use the same executable format. Sure, there are differences in things like the binary encodings of CPU instructions. The important thing is that the operating system loader and programming tools don't have to be completely rewritten for each new CPU that arrives on the scene.

The strength of Microsoft's commitment to get Windows NT up and running quickly is evidenced by the fact that they abandoned existing 32-bit tools and file formats. Virtual device drivers written for 16-bit Windows were using a different 32-bit file layout—the LE format—long before Windows NT appeared on the scene. More important than that is the shift of OBJ formats. Prior to the Windows NT C compiler, all Microsoft compilers used the Intel OMF (Object Module Format) specification. As mentioned earlier, the Microsoft compilers for Win32 produce COFF-format OBJ files. Some Microsoft competitors such as Borland and Symantec have chosen to forgo the COFF format OBJs and stick with the Intel OMF format. The upshot of this is that companies producing OBJs or LIBs for use with multiple compilers will need to go back to distributing separate versions of their products for different compilers (if they weren't already).

The PE format is documented (in the loosest sense of the word) in the WINNT.H header file. About midway through WINNT.H is a section titled "Image Format." This section starts out with small tidbits from the old familiar MS-DOS MZ format and NE format headers before moving into the newer PE information. WINNT.H provides definitions of the raw data structures used by PE files, but contains only a few useful comments to make sense of what the structures and flags mean. Whoever wrote the header file for the PE format (the name Michael J. O'Leary keeps popping up) is certainly a believer in long, descriptive names, along with deeply nested structures and macros. When coding with WINNT.H, it's not uncommon to have expressions like this:

```
pNTHeader->
OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;
```

To help make logical sense of the information in WINNT.H, read the Portable Executable and Common Object File Format Specification, available on MSDN Library quarterly CD-ROM releases up to and including October 2001.

Turning momentarily to the subject of COFF-format OBJs, the WINNT.H header file includes structure definitions and typedefs for COFF OBJ and LIB files. Unfortunately, I've been unable to find any documentation on this similar to that for the executable file mentioned above. Since PE files and COFF OBJ files are so similar, I decided that it was time to bring these files out into the light and document them as well.

Beyond just reading about what PE files are composed of, you'll also want to dump some PE files to see these concepts for yourself. If you use Microsoft® tools for Win32-based development, the DUMPBIN program will dissect and output PE files and COFF OBJ/LIB files in readable form. Of all the PE file dumpers, DUMPBIN is easily the most comprehensive. It even has a nifty option to disassemble the code sections in the file it's taking apart. Borland users can use TDUMP to view PE executable files, but TDUMP doesn't understand the COFF OBJ files. This isn't a big deal since the Borland compiler doesn't produce COFF-format OBJs in the first place.

I've written a PE and COFF OBJ file dumping program, PEDUMP (see Table 1), that I think provides more understandable output than DUMPBIN. Although it doesn't have a disassembler or work with LIB files, it is otherwise functionally equivalent to DUMPBIN, and adds a few new features to make it worth considering. The source code for PEDUMP is available on any MSJ bulletin board, so I won't list it here in its entirety. Instead, I'll show sample output from PEDUMP to illustrate the concepts as I describe them.

Table 1. PEDUMP.C

```
//--------------------
// PROGRAM: PEDUMP
// FILE:    PEDUMP.C
// AUTHOR:  Matt Pietrek - 1993
//--------------------
#include <windows.h>
#include <stdio.h>
#include "objdump.h"
#include "exedump.h"
#include "extrnvar.h"

// Global variables set here, and used in EXEDUMP.C and OBJDUMP.C
BOOL fShowRelocations = FALSE;
BOOL fShowRawSectionData = FALSE;
BOOL fShowSymbolTable = FALSE;
BOOL fShowLineNumbers = FALSE;

char HelpText[] =
"PEDUMP - Win32/COFF .EXE/.OBJ file dumper - 1993 Matt Pietrek\n\n"
"Syntax: PEDUMP [switches] filename\n\n"
"  /A    include everything in dump\n"
"  /H    include hex dump of sections\n"
"  /L    include line number information\n"
"  /R    show base relocations\n"
"  /S    show symbol table\n";

// Open up a file, memory map it, and call the appropriate dumping routine
void DumpFile(LPSTR filename)
{
    HANDLE hFile;
    HANDLE hFileMapping;
    LPVOID lpFileBase;
    PIMAGE_DOS_HEADER dosHeader;

    hFile = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ, NULL,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);

    if ( hFile = = INVALID_HANDLE_VALUE )
    {   printf("Couldn't open file with CreateFile()\n");
        return; }

    hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if ( hFileMapping = = 0 )
    {   CloseHandle(hFile);
        printf("Couldn't open file mapping with CreateFileMapping()\n");
        return; }

    lpFileBase = MapViewOfFile(hFileMapping, FILE_MAP_READ, 0, 0, 0);
    if ( lpFileBase = = 0 )
    {
        CloseHandle(hFileMapping);
        CloseHandle(hFile);
        printf("Couldn't map view of file with MapViewOfFile()\n");
        return;
    }
```

```
    printf("Dump of file %s\n\n", filename);

    dosHeader = (PIMAGE_DOS_HEADER)lpFileBase;
    if ( dosHeader->e_magic = = IMAGE_DOS_SIGNATURE )
       { DumpExeFile( dosHeader ); }
    else if ( (dosHeader->e_magic = = 0x014C)    // Does it look like a i386
              && (dosHeader->e_sp = = 0) )        // COFF OBJ file???
    {
        // The two tests above aren't what they look like.  They're
        // really checking for IMAGE_FILE_HEADER.Machine = = i386 (0x14C)
        // and IMAGE_FILE_HEADER.SizeOfOptionalHeader = = 0;
        DumpObjFile( (PIMAGE_FILE_HEADER)lpFileBase );
    }
    else
        printf("unrecognized file format\n");
    UnmapViewOfFile(lpFileBase);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);
}

// process all the command line arguments and return a pointer to
// the filename argument.
PSTR ProcessCommandLine(int argc, char *argv[])
{
    int i;

    for ( i=1; i < argc; i++ )
    {
        strupr(argv[i]);

        // Is it a switch character?
        if ( (argv[i][0] = = '-') || (argv[i][0] = = '/') )
        {
            if ( argv[i][1] = = 'A' )
            {   fShowRelocations = TRUE;
                fShowRawSectionData = TRUE;
                fShowSymbolTable = TRUE;
                fShowLineNumbers = TRUE; }
            else if ( argv[i][1] = = 'H' )
                fShowRawSectionData = TRUE;
            else if ( argv[i][1] = = 'L' )
                fShowLineNumbers = TRUE;
            else if ( argv[i][1] = = 'R' )
                fShowRelocations = TRUE;
            else if ( argv[i][1] = = 'S' )
                fShowSymbolTable = TRUE;
        }
        else    // Not a switch character.  Must be the filename
        {   return argv[i]; }
    }
}

int main(int argc, char *argv[])
{
    PSTR filename;

    if ( argc = = 1 )
```

```
    {   printf(    HelpText );
        return 1; }

    filename = ProcessCommandLine(argc, argv);
    if ( filename )
        DumpFile( filename );
    return 0;
}
```

Win32 and PE Basic Concepts

Let's go over a few fundamental ideas that permeate the design of a PE file (see Figure 1). I'll use the term "module" to mean the code, data, and resources of an executable file or DLL that have been loaded into memory. Besides code and data that your program uses directly, a module is also composed of the supporting data structures used by Windows to determine where the code and data is located in memory. In 16-bit Windows, the supporting data structures are in the module database (the segment referred to by an HMODULE). In Win32, these data structures are in the PE header, which I'll explain shortly.

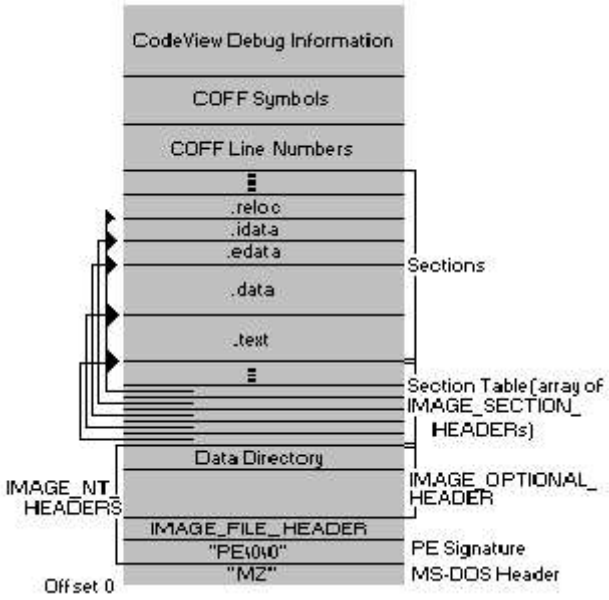

Figure 1. The PE file format

The first important thing to know about PE files is that the executable file on disk is very similar to what the module will look like after Windows has loaded it. The Windows loader doesn't need to work extremely hard to create a process from the disk file. The loader uses the memory-mapped file mechanism to map the appropriate pieces of the file into the virtual address space. To use a construction analogy, a PE file is like a prefabricated home. It's essentially brought into place in one piece, followed by a small amount of work to wire it up to the rest of the world (that is, to connect it to its DLLs and so on). This same ease of loading applies to PE-format DLLs as well. Once the module has been loaded, Windows can effectively treat it like any other memory-mapped file.

This is in marked contrast to the situation in 16-bit Windows. The 16-bit NE file loader reads in portions of the file and creates completely different data structures to represent the module in memory. When a code or data segment needs to be loaded, the loader has to allocate a new segment from the global heap, find where the raw data is stored in the executable file, seek to that location, read in the raw data, and apply any applicable fixups. In addition, each 16-bit module is responsible for remembering all the selectors it's currently using, whether the segment has been discarded, and so on.

For Win32, all the memory used by the module for code, data, resources, import tables, export tables, and other required module data structures is in one contiguous block of memory. All you need to know in this situation is where the loader mapped the file

into memory. You can easily find all the various pieces of the module by following pointers that are stored as part of the image.

Another idea you should be acquainted with is the Relative Virtual Address (RVA). Many fields in PE files are specified in terms of RVAs. An RVA is simply the offset of some item, relative to where the file is memory-mapped. For example, let's say the loader maps a PE file into memory starting at address 0x10000 in the virtual address space. If a certain table in the image starts at address 0x10464, then the table's RVA is 0x464.

```
(Virtual address 0x10464)-(base address 0x10000) = RVA 0x00464
```

To convert an RVA into a usable pointer, simply add the RVA to the base address of the module. The base address is the starting address of a memory-mapped EXE or DLL and is an important concept in Win32. For the sake of convenience, Windows NT and Windows 95 uses the base address of a module as the module's instance handle (HINSTANCE). In Win32, calling the base address of a module an HINSTANCE is somewhat confusing, because the term "instance handle" comes from 16-bit Windows. Each copy of an application in 16-bit Windows gets its own separate data segment (and an associated global handle) that distinguishes it from other copies of the application, hence the term instance handle. In Win32, applications don't need to be distinguished from one another because they don't share the same address space. Still, the term HINSTANCE persists to keep continuity between 16-bit Windows and Win32. What's important for Win32 is that you can call GetModuleHandle for any DLL that your process uses to get a pointer for accessing the module's components.

The final concept that you need to know about PE files is sections. A section in a PE file is roughly equivalent to a segment or the resources in an NE file. Sections contain either code or data. Unlike segments, sections are blocks of contiguous memory with no size constraints. Some sections contain code or data that your program declared and uses directly, while other data sections are created for you by the linker and librarian, and contain information vital to the operating system. In some descriptions of the PE format, sections are also referred to as objects. The term object has so many overloaded meanings that I'll stick to calling the code and data areas sections.

The PE Header

Like all other executable file formats, the PE file has a collection of fields at a known (or easy to find) location that define what the rest of the file looks like. This header contains information such as the locations and sizes of the code and data areas, what operating system the file is intended for, the initial stack size, and other vital pieces of information that I'll discuss shortly. As with other executable formats from Microsoft, this main header isn't at the very beginning of the file. The first few hundred bytes of the typical PE file are taken up by the MS-DOS stub. This stub is a tiny program that prints out something to the effect of "This program cannot be run in MS-DOS mode." So if you run a Win32-based program in an environment that doesn't support Win32, you'll get this informative error message. When the Win32 loader memory maps a PE file, the first byte of the mapped file corresponds to the first byte of the MS-DOS stub. That's right. With every Win32-based program you start up, you get an MS-DOS-based program loaded for free!

As in other Microsoft executable formats, you find the real header by looking up its starting offset, which is stored in the MS-DOS stub header. The WINNT.H file includes a structure definition for the MS-DOS stub header that makes it very easy to look up where the PE header starts. The e_lfanew field is a relative offset (or RVA, if you prefer) to the actual PE header. To get a pointer to the PE header in memory, just add that field's value to the image base:

```
// Ignoring typecasts and pointer conversion issues for clarity...
pNTHeader = dosHeader + dosHeader->e_lfanew;
```

Once you have a pointer to the main PE header, the fun can begin. The main PE header is a structure of type IMAGE_NT_HEADERS, which is defined in WINNT.H. This structure is composed of a DWORD and two substructures and is laid

out as follows:

```
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADER OptionalHeader;
```

The Signature field viewed as ASCII text is "PE\0\0". If after using the e_lfanew field in the MS-DOS header, you find an NE signature here rather than a PE, you're working with a 16-bit Windows NE file. Likewise, an LE in the signature field would indicate a Windows 3.x virtual device driver (VxD). An LX here would be the mark of a file for OS/2 2.0.

Following the PE signature DWORD in the PE header is a structure of type IMAGE_FILE_HEADER. The fields of this structure contain only the most basic information about the file. The structure appears to be unmodified from its original COFF implementations. Besides being part of the PE header, it also appears at the very beginning of the COFF OBJs produced by the Microsoft Win32 compilers. The fields of the IMAGE_FILE_HEADER are shown in Table 2.

Table 2. IMAGE_FILE_HEADER Fields

`WORD Machine`
: The CPU that this file is intended for. The following CPU IDs are defined:

0x14d	Intel i860
0x14c	Intel I386 (same ID used for 486 and 586)
0x162	MIPS R3000
0x166	MIPS R4000
0x183	DEC Alpha AXP

`WORD NumberOfSections`
: The number of sections in the file.

`DWORD TimeDateStamp`
: The time that the linker (or compiler for an OBJ file) produced this file. This field holds the number of seconds since December 31st, 1969, at 4:00 P.M.

`DWORD PointerToSymbolTable`
: The file offset of the COFF symbol table. This field is only used in OBJ files and PE files with COFF debug information. PE files support multiple debug formats, so debuggers should refer to the IMAGE_DIRECTORY_ENTRY_DEBUG entry in the data directory (defined later).

`DWORD NumberOfSymbols`
: The number of symbols in the COFF symbol table. See above.

`WORD SizeOfOptionalHeader`
: The size of an optional header that can follow this structure. In OBJs, the field is 0. In executables, it is the size of the IMAGE_OPTIONAL_HEADER structure that follows this structure.

`WORD Characteristics`
: Flags with information about the file. Some important fields:

0x0001	There are no relocations in this file
0x0002	File is an executable image (not a OBJ or LIB)
0x2000	File is a dynamic-link library, not a program

Other fields are defined in WINNT.H

The third component of the PE header is a structure of type IMAGE_OPTIONAL_HEADER. For PE files, this portion certainly isn't optional. The COFF format allows individual implementations to define a structure of additional information beyond the standard IMAGE_FILE_HEADER. The fields in the IMAGE_OPTIONAL_HEADER are what the PE designers felt was critical information beyond the basic information in the IMAGE_FILE_HEADER.

All of the fields of the IMAGE_OPTIONAL_HEADER aren't necessarily important to know about (see Figure 4). The more important ones to be aware of are the ImageBase and the Subsystem fields. You can skim or skip the description of the fields.

Table 3. IMAGE_OPTIONAL_HEADER Fields

`WORD Magic`
: Appears to be a signature WORD of some sort. Always appears to be set to 0x010B.

`BYTE MajorLinkerVersion`

`BYTE MinorLinkerVersion`
: The version of the linker that produced this file. The numbers should be displayed as decimal values, rather than as hex. A typical linker version is 2.23.

`DWORD SizeOfCode`
: The combined and rounded-up size of all the code sections. Usually, most files only have one code section, so this field matches the size of the .text section.

`DWORD SizeOfInitializedData`
: This is supposedly the total size of all the sections that are composed of initialized data (not including code segments.) However, it doesn't seem to be consistent with what appears in the file.

`DWORD SizeOfUninitializedData`
: The size of the sections that the loader commits space for in the virtual address space, but that don't take up any space in the disk file. These sections don't need to have specific values at program startup, hence the term uninitialized data. Uninitialized data usually goes into a section called .bss.

`DWORD AddressOfEntryPoint`
: The address where the loader will begin execution. This is an RVA, and usually can usually be found in the .text section.

`DWORD BaseOfCode`
: The RVA where the file's code sections begin. The code sections typically come before the data sections and after the PE header in memory. This RVA is usually 0x1000 in Microsoft Linker-produced EXEs. Borland's TLINK32 looks like it adds the image base to the RVA of the first code section and stores the result in this field.

`DWORD BaseOfData`
: The RVA where the file's data sections begin. The data sections typically come last in memory, after the PE header and the code sections.

`DWORD ImageBase`
: When the linker creates an executable, it assumes that the file will be memory-mapped to a specific location in memory. That address is stored in this field, assuming a load address allows linker optimizations to take place. If the file really is memory-mapped to that address by the loader, the code doesn't need any patching before it can be run. In executables produced for Windows NT, the default image base is 0x10000. For DLLs, the default is 0x400000. In Windows 95, the address 0x10000 can't be used to load 32-bit EXEs because it lies within a linear address region shared by all processes. Because of this, Microsoft has changed the default base address for Win32 executables to 0x400000. Older programs that

were linked assuming a base address of 0x10000 will take longer to load under Windows 95 because the loader needs to apply the base relocations.

`DWORD SectionAlignment`

When mapped into memory, each section is guaranteed to start at a virtual address that's a multiple of this value. For paging purposes, the default section alignment is 0x1000.

`DWORD FileAlignment`

In the PE file, the raw data that comprises each section is guaranteed to start at a multiple of this value. The default value is 0x200 bytes, probably to ensure that sections always start at the beginning of a disk sector (which are also 0x200 bytes in length). This field is equivalent to the segment/resource alignment size in NE files. Unlike NE files, PE files typically don't have hundreds of sections, so the space wasted by aligning the file sections is almost always very small.

`WORD MajorOperatingSystemVersion`

`WORD MinorOperatingSystemVersion`

The minimum version of the operating system required to use this executable. This field is somewhat ambiguous since the subsystem fields (a few fields later) appear to serve a similar purpose. This field defaults to 1.0 in all Win32 EXEs to date.

`WORD MajorImageVersion`

`WORD MinorImageVersion`

A user-definable field. This allows you to have different versions of an EXE or DLL. You set these fields via the linker /VERSION switch. For example, "LINK /VERSION:2.0 myobj.obj".

`WORD MajorSubsystemVersion`

`WORD MinorSubsystemVersion`

Contains the minimum subsystem version required to run the executable. A typical value for this field is 3.10 (meaning Windows NT 3.1).

`DWORD Reserved1`

Seems to always be 0.

`DWORD SizeOfImage`

This appears to be the total size of the portions of the image that the loader has to worry about. It is the size of the region starting at the image base up to the end of the last section. The end of the last section is rounded up to the nearest multiple of the section alignment.

`DWORD SizeOfHeaders`

The size of the PE header and the section (object) table. The raw data for the sections starts immediately after all the header components.

`DWORD CheckSum`

Supposedly a CRC checksum of the file. As in other Microsoft executable formats, this field is ignored and set to 0. The one exception to this rule is for trusted services and these EXEs must have a valid checksum.

`WORD Subsystem`

The type of subsystem that this executable uses for its user interface. WINNT.H defines the following values:

NATIVE	1	Doesn't require a subsystem (such as a device driver)
WINDOWS_GUI	2	Runs in the Windows GUI subsystem
WINDOWS_CUI	3	Runs in the Windows character subsystem (a console app)
OS2_CUI	5	Runs in the OS/2 character subsystem (OS/2 1.x apps only)
POSIX_CUI	7	Runs in the Posix character subsystem

`WORD DllCharacteristics`

A set of flags indicating under which circumstances a DLL's initialization function (such as DllMain) will be called. This value appears to always be set to 0, yet the operating system still calls the DLL initialization function for all four events.

The following values are defined:

1	Call when DLL is first loaded into a process's address space
2	Call when a thread terminates
4	Call when a thread starts up
8	Call when DLL exits

`DWORD SizeOfStackReserve`

The amount of virtual memory to reserve for the initial thread's stack. Not all of this memory is committed, however (see the next field). This field defaults to 0x100000 (1MB). If you specify 0 as the stack size to CreateThread, the resulting thread will also have a stack of this same size.

`DWORD SizeOfStackCommit`

The amount of memory initially committed for the initial thread's stack. This field defaults to 0x1000 bytes (1 page) for the Microsoft Linker while TLINK32 makes it two pages.

`DWORD SizeOfHeapReserve`

The amount of virtual memory to reserve for the initial process heap. This heap's handle can be obtained by calling GetProcessHeap. Not all of this memory is committed (see the next field).

`DWORD SizeOfHeapCommit`

The amount of memory initially committed in the process heap. The default is one page.

`DWORD LoaderFlags`

From WINNT.H, these appear to be fields related to debugging support. I've never seen an executable with either of these bits enabled, nor is it clear how to get the linker to set them. The following values are defined:

1.	Invoke a breakpoint instruction before starting the process
2.	Invoke a debugger on the process after it's been loaded

`DWORD NumberOfRvaAndSizes`

The number of entries in the DataDirectory array (below). This value is always set to 16 by the current tools.

`IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]`

An array of IMAGE_DATA_DIRECTORY structures. The initial array elements contain the starting RVA and sizes of important portions of the executable file. Some elements at the end of the array are currently unused. The first element of the array is always the address and size of the exported function table (if present). The second array entry is the address and size of the imported function table, and so on. For a complete list of defined array entries, see the IMAGE_DIRECTORY_ENTRY_XXX #defines in WINNT.H. This array allows the loader to quickly find a particular section of the image (for example, the imported function table), without needing to iterate through each of the images sections, comparing names as it goes along. Most array entries describe an entire section's data. However, the IMAGE_DIRECTORY_ENTRY_DEBUG element only encompasses a small portion of the bytes in the .rdata section.

The Section Table

Between the PE header and the raw data for the image's sections lies the section table. The section table is essentially a phone book containing information about each section in the image. The sections in the image are sorted by their starting address (RVAs), rather than alphabetically.

Now I can better clarify what a section is. In an NE file, your program's code and data are stored in distinct "segments" in the file. Part of the NE header is an array of structures, one for each segment your program uses. Each structure in the array contains information about one segment. The information stored includes the segment's type (code or data), its size, and its location elsewhere in the file. In a PE file, the section table is analogous to the segment table in the NE file. Unlike an NE file segment table, though, a PE section table doesn't store a selector value for each code or data chunk. Instead, each section table entry stores an address where the file's raw data has been mapped into memory. While sections are analogous to 32-bit segments, they really aren't individual segments. They're just really memory ranges in a process's virtual address space.

Another area where PE files differ from NE files is how they manage the supporting data that your program doesn't use, but the operating system does; for example, the list of DLLs that the executable uses or the location of the fixup table. In an NE file, resources aren't considered segments. Even though they have selectors assigned to them, information about resources is not stored in the NE header's segment table. Instead, resources are relegated to a separate table towards the end of the NE header. Information about imported and exported functions also doesn't warrant its own segment; it's crammed into the NE header.

The story with PE files is different. Anything that might be considered vital code or data is stored in a full-fledged section. Thus, information about imported functions is stored in its own section, as is the table of functions that the module exports. The same goes for the relocation data. Any code or data that might be needed by either the program or the operating system gets its own section.

Before I discuss specific sections, I need to describe the data that the operating system manages the sections with. Immediately following the PE header in memory is an array of IMAGE_SECTION_HEADERs. The number of elements in this array is given in the PE header (the IMAGE_NT_HEADER.FileHeader.NumberOfSections field). I used PEDUMP to output the section table and all of the section's fields and attributes. Figure 5 shows the PEDUMP output of a section table for a typical EXE file, and Figure 6 shows the section table in an OBJ file.

Table 4. A Typical Section Table from an EXE File

```
01 .text     VirtSize: 00005AFA  VirtAddr:  00001000
    raw data offs:   00000400  raw data size: 00005C00
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00009220  line #'s:      0000020C
    characteristics: 60000020
      CODE  MEM_EXECUTE  MEM_READ

  02 .bss      VirtSize: 00001438  VirtAddr:  00007000
    raw data offs:   00000000  raw data size: 00001600
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: C0000080
      UNINITIALIZED_DATA  MEM_READ  MEM_WRITE

  03 .rdata    VirtSize: 0000015C  VirtAddr:  00009000
    raw data offs:   00006000  raw data size: 00000200
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 40000040
      INITIALIZED_DATA  MEM_READ
```

```
  04 .data     VirtSize: 0000239C  VirtAddr:  0000A000
    raw data offs:   00006200  raw data size: 00002400
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  05 .idata    VirtSize: 0000033E  VirtAddr:  0000D000
    raw data offs:   00008600  raw data size: 00000400
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: C0000040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  06 .reloc    VirtSize: 000006CE  VirtAddr:  0000E000
    raw data offs:   00008A00  raw data size: 00000800
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 42000040
      INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

Table 5. A Typical Section Table from an OBJ File

```
01 .drectve  PhysAddr: 00000000  VirtAddr:  00000000
    raw data offs:   000000DC  raw data size: 00000026
    relocation offs: 00000000  relocations:   00000000
    line # offs:     00000000  line #'s:      00000000
    characteristics: 00100A00
      LNK_INFO  LNK_REMOVE

  02 .debug$S  PhysAddr: 00000026  VirtAddr:  00000000
    raw data offs:   00000102  raw data size: 000016D0
    relocation offs: 000017D2  relocations:   00000032
    line # offs:     00000000  line #'s:      00000000
    characteristics: 42100048
      INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ

  03 .data     PhysAddr: 000016F6  VirtAddr:  00000000
    raw data offs:   000019C6  raw data size: 00000D87
    relocation offs: 0000274D  relocations:   00000045
    line # offs:     00000000  line #'s:      00000000
    characteristics: C0400040
      INITIALIZED_DATA  MEM_READ  MEM_WRITE

  04 .text     PhysAddr: 0000247D  VirtAddr:  00000000
    raw data offs:   000029FF  raw data size: 000010DA
    relocation offs: 00003AD9  relocations:   000000E9
    line # offs:     000043F3  line #'s:      000000D9
    characteristics: 60500020
      CODE  MEM_EXECUTE  MEM_READ

  05 .debug$T  PhysAddr: 00003557  VirtAddr:  00000000
    raw data offs:   00004909  raw data size: 00000030
```

```
relocation offs: 00000000  relocations:   00000000
line # offs:     00000000  line #'s:      00000000
characteristics: 42100048
  INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

Each IMAGE_SECTION_HEADER has the format described in Figure 7. It's interesting to note what's missing from the information stored for each section. First off, notice that there's no indication of any PRELOAD attributes. The NE file format allows you to specify with the PRELOAD attribute which segments should be loaded at module load time. The OS/2® 2.0 LX format has something similar, allowing you to specify up to eight pages to preload. The PE format has nothing like this. Microsoft must be confident in the performance of Win32 demand-paged loading.

Table 6. IMAGE_SECTION_HEADER Formats

`BYTE Name[IMAGE_SIZEOF_SHORT_NAME]`
: This is an 8-byte ANSI name (not UNICODE) that names the section. Most section names start with a . (such as ".text"), but this is not a requirement, as some PE documentation would have you believe. You can name your own sections with either the segment directive in assembly language, or with "#pragma data_seg" and "#pragma code_seg" in the Microsoft C/C++ compiler. It's important to note that if the section name takes up the full 8 bytes, there's no NULL terminator byte. If you're a printf devotee, you can use %.8s to avoid copying the name string to another buffer where you can NULL-terminate it.

```
union {
DWORD PhysicalAddress
DWORD VirtualSize
} Misc;
```

: This field has different meanings, in EXEs or OBJs. In an EXE, it holds the actual size of the code or data. This is the size before rounding up to the nearest file alignment multiple. The SizeOfRawData field (seems a bit of a misnomer) later on in the structure holds the rounded up value. The Borland linker reverses the meaning of these two fields and appears to be correct. For OBJ files, this field indicates the physical address of the section. The first section starts at address 0. To find the physical address in an OBJ file of the next section, add the SizeOfRawData value to the physical address of the current section.

`DWORD VirtualAddress`
: In EXEs, this field holds the RVA to where the loader should map the section. To calculate the real starting address of a given section in memory, add the base address of the image to the section's VirtualAddress stored in this field. With Microsoft tools, the first section defaults to an RVA of 0x1000. In OBJs, this field is meaningless and is set to 0.

`DWORD SizeOfRawData`
: In EXEs, this field contains the size of the section after it's been rounded up to the file alignment size. For example, assume a file alignment size of 0x200. If the VirtualSize field from above says that the section is 0x35A bytes in length, this field will say that the section is 0x400 bytes long. In OBJs, this field contains the exact size of the section emitted by the compiler or assembler. In other words, for OBJs, it's equivalent to the VirtualSize field in EXEs.

`DWORD PointerToRawData`
: This is the file-based offset of where the raw data emitted by the compiler or assembler can be found. If your program memory maps a PE or COFF file itself (rather than letting the operating system load it), this field is more important than the VirtualAddress field. You'll have a completely linear file mapping in this situation, so you'll find the data for the sections at this offset, rather than at the RVA specified in the VirtualAddress field.

`DWORD PointerToRelocations`
: In OBJs, this is the file-based offset to the relocation information for this section. The relocation information for each OBJ section immediately follows the raw data for that section. In EXEs, this field (and the subsequent field) are meaningless, and set to 0. When the linker creates the EXE, it resolves most of the fixups, leaving only base address relocations and imported functions to be resolved at load time. The information about base relocations and imported functions is kept in their own sections, so there's no need for an EXE to have per-section relocation data following the raw section data.

`DWORD PointerToLinenumbers`
: This is the file-based offset of the line number table. A line number table correlates source file line numbers to the addresses of the code generated for a given line. In modern debug formats like the CodeView format, line number information is stored as part of the debug information. In the COFF debug format, however, the line number information

is stored separately from the symbolic name/type information. Usually, only code sections (such as .text) have line numbers. In EXE files, the line numbers are collected towards the end of the file, after the raw data for the sections. In OBJ files, the line number table for a section comes after the raw section data and the relocation table for that section.

`WORD NumberOfRelocations`

The number of relocations in the relocation table for this section (the PointerToRelocations field from above). This field seems relevant only for OBJ files.

`WORD NumberOfLinenumbers`

The number of line numbers in the line number table for this section (the PointerToLinenumbers field from above).

`DWORD Characteristics`

What most programmers call flags, the COFF/PE format calls characteristics. This field is a set of flags that indicate the section's attributes (such as code/data, readable, or writeable,). For a complete list of all possible section attributes, see the IMAGE_SCN_XXX_XXX #defines in WINNT.H. Some of the more important flags are shown below:

0x00000020 This section contains code. Usually set in conjunction with the executable flag (0x80000000).

0x00000040 This section contains initialized data. Almost all sections except executable and the .bss section have this flag set.

0x00000080 This section contains uninitialized data (for example, the .bss section).

0x00000200 This section contains comments or some other type of information. A typical use of this section is the .drectve section emitted by the compiler, which contains commands for the linker.

0x00000800 This section's contents shouldn't be put in the final EXE file. These sections are used by the compiler/assembler to pass information to the linker.

0x02000000 This section can be discarded, since it's not needed by the process once it's been loaded. The most common discardable section is the base relocations (.reloc).

0x10000000 This section is shareable. When used with a DLL, the data in this section will be shared among all processes using the DLL. The default is for data sections to be nonshared, meaning that each process using a DLL gets its own copy of this section's data. In more technical terms, a shared section tells the memory manager to set the page mappings for this section such that all processes using the DLL refer to the same physical page in memory. To make a section shareable, use the SHARED attribute at link time. For example

```
LINK /SECTION:MYDATA,RWS ...
```

tells the linker that the section called MYDATA should be readable, writeable, and shared.

0x20000000 This section is executable. This flag is usually set whenever the "contains code" flag (0x00000020) is set.

0x40000000 This section is readable. This flag is almost always set for sections in EXE files.

0x80000000 The section is writeable. If this flag isn't set in an EXE's section, the loader should mark the memory mapped pages as read-only or execute-only. Typical sections with this attribute are .data and .bss. Interestingly, the .idata section also has this attribute set.

Also missing from the PE format is the notion of page tables. The OS/2 equivalent of an IMAGE_SECTION_HEADER in the LX format doesn't point directly to where the code or data for a section can be found in the file. Instead, it refers to a page lookup table that specifies attributes and the locations of specific ranges of pages within a section. The PE format dispenses with all that, and guarantees that a section's data will be stored contiguously within the file. Of the two formats, the LX method may allow more flexibility, but the PE style is significantly simpler and easier to work with. Having written file dumpers for both formats, I can vouch for this!

Another welcome change in the PE format is that the locations of items are stored as simple DWORD offsets. In the NE format, the location of almost everything is stored as a sector value. To find the real offset, you need to first look up the alignment unit size in the NE header and convert it to a sector size (typically 16 or 512 bytes). You then need to multiply the sector size by the specified sector offset to get an actual file offset. If by chance something isn't stored as a sector offset in an NE file, it is probably stored as an offset relative to the NE header. Since the NE header isn't at the beginning of the file, you need to drag around the file offset of the NE header in your code. All in all, the PE format is much easier to work with than the NE, LX, or LE formats (assuming you can use memory-mapped files).

Common Sections

Having seen what sections are in general and where they're located, let's look at the common sections that you'll find in EXE and OBJ files. The list is by no means complete, but includes the sections you encounter every day (even if you're not aware of it).

The .text section is where all general-purpose code emitted by the compiler or assembler ends up. Since PE files run in 32-bit mode and aren't restricted to 16-bit segments, there's no reason to break the code from separate source files into separate sections. Instead, the linker concatenates all the .text sections from the various OBJs into one big .text section in the EXE. If you use Borland C++ the compiler emits its code to a segment named CODE. PE files produced with Borland C++ have a section named CODE rather than one called .text. I'll explain this in a minute.

It was somewhat interesting to me to find out that there was additional code in the .text section beyond what I created with the compiler or used from the run-time libraries. In a PE file, when you call a function in another module (for example, GetMessage in USER32.DLL), the CALL instruction emitted by the compiler doesn't transfer control directly to the function in the DLL (see Figure 8). Instead, the call instruction transfers control to a

```
JMP DWORD PTR [XXXXXXXX]
```

instruction that's also in the .text section. The JMP instruction indirects through a DWORD variable in the .idata section. This .idata section DWORD contains the real address of the operating system function entry point. After thinking about this for a while, I came to understand why DLL calls are implemented this way. By funneling all calls to a given DLL function through one location, the loader doesn't need to patch every instruction that calls a DLL. All the PE loader has to do is put the correct address of the target function into the DWORD in the .idata section. No call instructions need to be patched. This is in marked contrast to NE files, where each segment contains a list of fixups that need to be applied to the segment. If the segment calls a given DLL function 20 times, the loader must write the address of that function 20 times into the segment. The downside to the PE method is that you can't initialize a variable with the true address of a DLL function. For example, you would think that something like

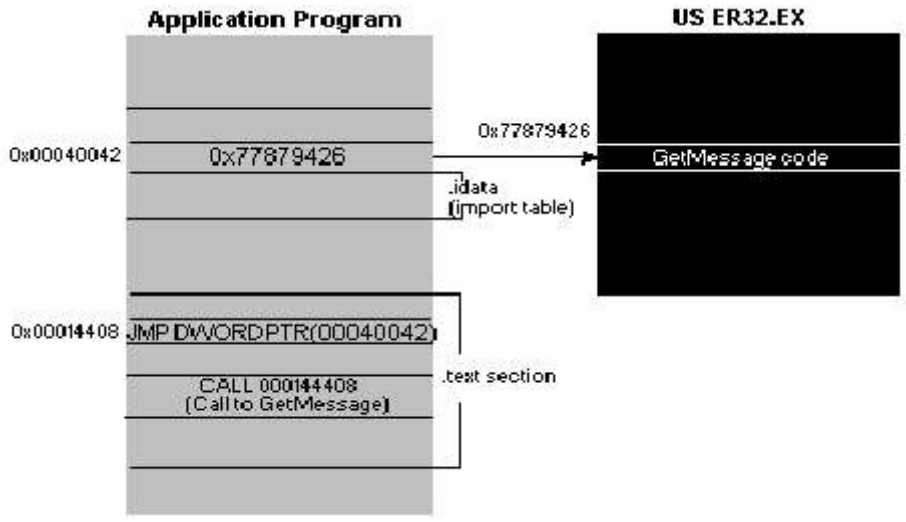

Figure 2. Calling a function in another module

```
FARPROC pfnGetMessage = GetMessage;
```

would put the address of GetMessage into the variable pfnGetMessage. In 16-bit Windows, this works, while in Win32 it doesn't. In Win32, the variable pfnGetMessage will end up holding the address of the JMP DWORD PTR [XXXXXXXX] thunk that I mentioned earlier. If you wanted to call through the function pointer, things would work as you'd expect. However, if you want to read the bytes at the beginning of GetMessage, you're out of luck (unless you do additional work to follow the .idata "pointer" yourself). I'll come back to this topic later, in the discussion of the import table.

Although Borland could have had the compiler emit segments with a name of .text, it chose a default segment name of CODE. To determine a section name in the PE file, the Borland linker (TLINK32.EXE) takes the segment name from the OBJ file and truncates it to 8 characters (if necessary).

While the difference in the section names is a small matter, there is a more important difference in how Borland PE files link to other modules. As I mentioned in the .text description, all calls to OBJs go through a JMP DWORD PTR [XXXXXXXX] thunk. Under the Microsoft system, this thunk comes to the EXE from the .text section of an import library. Because the library manager (LIB32) creates the import library (and the thunk) when you link the external DLL, the linker doesn't have to "know" how to generate these thunks itself. The import library is really just some more code and data to link into the PE file.

The Borland system of dealing with imported functions is simply an extension of the way things were done for 16-bit NE files. The import libraries that the Borland linker uses are really just a list of function names along with the name of the DLL they're in. TLINK32 is therefore responsible for determining which fixups are to external DLLs, and generating an appropriate JMP DWORD PTR [XXXXXXXX] thunk for it. TLINK32 stores the thunks that it creates in a section named .icode.

Just as .text is the default section for code, the .data section is where your initialized data goes. This data consists of global and static variables that are initialized at compile time. It also includes string literals. The linker combines all the .data sections from the OBJ and LIB files into one .data section in the EXE. Local variables are located on a thread's stack, and take no room in the .data or .bss sections.

The .bss section is where any uninitialized static and global variables are stored. The linker combines all the .bss sections in the OBJ and LIB files into one .bss section in the EXE. In the section table, the RawDataOffset field for the .bss section is set to 0, indicating that this section doesn't take up any space in the file. TLINK doesn't emit this section. Instead it extends the virtual size of the DATA section.

.CRT is another initialized data section utilized by the Microsoft C/C++ run-time libraries (hence the name). Why this data couldn't go into the standard .data section is beyond me.

The .rsrc section contains all the resources for the module. In the early days of Windows NT, the RES file output of the 16-bit RC.EXE wasn't in a format that the Microsoft PE linker could understand. The CVTRES program converted these RES files into a COFF-format OBJ, placing the resource data into a .rsrc section within the OBJ. The linker could then treat the resource OBJ as just another OBJ to link in, allowing the linker to not "know" anything special about resources. More recent linkers from Microsoft appear to be able to process RES files directly.

The .idata section contains information about functions (and data) that the module imports from other DLLs. This section is equivalent to an NE file's module reference table. A key difference is that each function that a PE file imports is specifically listed in this section. To find the equivalent information in an NE file, you'd have to go digging through the relocations at the end of the raw data for each of the segments.

The .edata section is a list of the functions and data that the PE file exports for other modules. Its NE file equivalent is the combination of the entry table, the resident names table, and the nonresident names table. Unlike in 16-bit Windows, there's seldom a reason to export anything from an EXE file, so you usually only see .edata sections in DLLs. When using Microsoft tools,

the data in the .edata section comes to the PE file via the EXP file. Put another way, the linker doesn't generate this information on its own. Instead, it relies on the library manager (LIB32) to scan the OBJ files and create the EXP file that the linker adds to its list of modules to link. Yes, that's right! Those pesky EXP files are really just OBJ files with a different extension.

The .reloc section holds a table of base relocations. A base relocation is an adjustment to an instruction or initialized variable value that's needed if the loader couldn't load the file where the linker assumed it would. If the loader is able to load the image at the linker's preferred base address, the loader completely ignores the relocation information in this section. If you want to take a chance and hope that the loader can always load the image at the assumed base address, you can tell the linker to strip this information with the /FIXED option. While this may save space in the executable file, it may cause the executable not to work on other Win32-based implementations. For example, say you built an EXE for Windows NT and based the EXE at 0x10000. If you told the linker to strip the relocations, the EXE wouldn't run under Windows 95, where the address 0x10000 is already in use.

It's important to note that the JMP and CALL instructions that the compiler generates use offsets relative to the instruction, rather than actual offsets in the 32-bit flat segment. If the image needs to be loaded somewhere other than where the linker assumed for a base address, these instructions don't need to change, since they use relative addressing. As a result, there are not as many relocations as you might think. Relocations are usually only needed for instructions that use a 32-bit offset to some data. For example, let's say you had the following global variable declarations:

```
int i;
int *ptr = &i;
```

If the linker assumed an image base of 0x10000, the address of the variable i will end up containing something like 0x12004. At the memory used to hold the pointer "ptr", the linker will have written out 0x12004, since that's the address of the variable i. If the loader for whatever reason decided to load the file at a base address of 0x70000, the address of i would be 0x72004. The .reloc section is a list of places in the image where the difference between the linker assumed load address and the actual load address needs to be factored in.

When you use the compiler directive _ _declspec(thread), the data that you define doesn't go into either the .data or .bss sections. It ends up in the .tls section, which refers to "thread local storage," and is related to the TlsAlloc family of Win32 functions. When dealing with a .tls section, the memory manager sets up the page tables so that whenever a process switches threads, a new set of physical memory pages is mapped to the .tls section's address space. This permits per-thread global variables. In most cases, it is much easier to use this mechanism than to allocate memory on a per-thread basis and store its pointer in a TlsAlloc'ed slot.

There's one unfortunate note that must be added about the .tls section and _ _declspec(thread) variables. In Windows NT and Windows 95, this thread local storage mechanism won't work in a DLL if the DLL is loaded dynamically by LoadLibrary. In an EXE or an implicitly loaded DLL, everything works fine. If you can't implicitly link to the DLL, but need per-thread data, you'll have to fall back to using TlsAlloc and TlsGetValue with dynamically allocated memory.

Although the .rdata section usually falls between the .data and .bss sections, your program generally doesn't see or use the data in this section. The .rdata section is used for at least two things. First, in Microsoft linker-produced EXEs, the .rdata section holds the debug directory, which is only present in EXE files. (In TLINK32 EXEs, the debug directory is in a section named .debug.) The debug directory is an array of IMAGE_DEBUG_DIRECTORY structures. These structures hold information about the type, size, and location of the various types of debug information stored in the file. Three main types of debug information appear: CodeView®, COFF, and FPO. Figure 9 shows the PEDUMP output for a typical debug directory.

Table 7. A Typical Debug Directory

Type	Size	Address	FilePtr	Charactr	TimeData	Version	

COFF	000065C5	00000000	00009200	00000000	2CF8CF3D		0.00
???	00000114	00000000	0000F7C8	00000000	2CF8CF3D		0.00
FPO	000004B0	00000000	0000F8DC	00000000	2CF8CF3D		0.00
CODEVIEW	0000B0B4	00000000	0000FD8C	00000000	2CF8CF3D		0.00

The debug directory isn't necessarily found at the beginning of the .rdata section. To find the start of the debug directory table, use the RVA in the seventh entry (IMAGE_DIRECTORY_ENTRY_DEBUG) of the data directory. The data directory is at the end of the PE header portion of the file. To determine the number of entries in the Microsoft linker-generated debug directory, divide the size of the debug directory (found in the size field of the data directory entry) by the size of an IMAGE_DEBUG_DIRECTORY structure. TLINK32 emits a simple count, usually 1. The PEDUMP sample program demonstrates this.

The other useful portion of an .rdata section is the description string. If you specified a DESCRIPTION entry in your program's DEF file, the specified description string appears in the .rdata section. In the NE format, the description string is always the first entry of the nonresident names table. The description string is intended to hold a useful text string describing the file. Unfortunately, I haven't found an easy way to find it. I've seen PE files that had the description string before the debug directory, and other files that had it after the debug directory. I'm not aware of any consistent method of finding the description string (or even if it's present at all).

These .debug$S and .debug$T sections only appear in OBJs. They store the CodeView symbol and type information. The section names are derived from the segment names used for this purpose by previous 16-bit compilers ($$SYMBOLS and $$TYPES). The sole purpose of the .debug$T section is to hold the pathname to the PDB file that contains the CodeView information for all the OBJs in the project. The linker reads in the PDB and uses it to create portions of the CodeView information that it places at the end of the finished PE file.

The .drective section only appears in OBJ files. It contains text representations of commands for the linker. For example, in any OBJ I compile with the Microsoft compiler, the following strings appear in the .drectve section:

```
-defaultlib:LIBC -defaultlib:OLDNAMES
```

When you use _ _declspec(export) in your code, the compiler simply emits the command-line equivalent into the .drectve section (for instance, "-export:MyFunction").

In playing around with PEDUMP, I've encountered other sections from time to time. For instance, in the Windows 95 KERNEL32.DLL, there are LOCKCODE and LOCKDATA sections. Presumably these are sections that will get special paging treatment so that they're never paged out of memory.

There are two lessons to be learned from this. First, don't feel constrained to use only the standard sections provided by the compiler or assembler. If you need a separate section for some reason, don't hesitate to create your own. In the C/C++ compiler, use the #pragma code_seg and #pragma data_seg. In assembly language, just create a 32-bit segment (which becomes a section) with a name different from the standard sections. If using TLINK32, you must use a different class or turn off code segment packing. The other thing to remember is that section names that are out of the ordinary can often give a deeper insight into the purpose and implementation of a particular PE file.

PE File Imports

Earlier, I described how function calls to outside DLLs don't call the DLL directly. Instead, the CALL instruction goes to a JMP DWORD PTR [XXXXXXXX] instruction somewhere in the executable's .text section (or .icode section if you're using Borland C++).

The address that the JMP instruction looks up and transfers control to is the real target address. The PE file's .idata section contains the information necessary for the loader to determine the addresses of the target functions and patch them into the executable image.

The .idata section (or import table, as I prefer to call it) begins with an array of IMAGE_IMPORT_DESCRIPTORs. There is one IMAGE_IMPORT_DESCRIPTOR for each DLL that the PE file implicitly links to. There's no field indicating the number of structures in this array. Instead, the last element of the array is indicated by an IMAGE_IMPORT_DESCRIPTOR that has fields filled with NULLs. The format of an IMAGE_IMPORT_DESCRIPTOR is shown in Figure 10.

Table 8. IMAGE_IMPORT_DESCRIPTOR Format

`DWORD Characteristics`
: At one time, this may have been a set of flags. However, Microsoft changed its meaning and never bothered to update WINNT.H. This field is really an offset (an RVA) to an array of pointers. Each of these pointers points to an IMAGE_IMPORT_BY_NAME structure.

`DWORD TimeDateStamp`
: The time/date stamp indicating when the file was built.

`DWORD ForwarderChain`
: This field relates to forwarding. Forwarding involves one DLL sending on references to one of its functions to another DLL. For example, in Windows NT, NTDLL.DLL appears to forward some of its exported functions to KERNEL32.DLL. An application may think it's calling a function in NTDLL.DLL, but it actually ends up calling into KERNEL32.DLL. This field contains an index into FirstThunk array (described momentarily). The function indexed by this field will be forwarded to another DLL. Unfortunately, the format of how a function is forwarded isn't documented, and examples of forwarded functions are hard to find.

`DWORD Name`
: This is an RVA to a NULL-terminated ASCII string containing the imported DLL's name. Common examples are "KERNEL32.DLL" and "USER32.DLL".

`PIMAGE_THUNK_DATA FirstThunk`
: This field is an offset (an RVA) to an IMAGE_THUNK_DATA union. In almost every case, the union is interpreted as a pointer to an IMAGE_IMPORT_BY_NAME structure. If the field isn't one of these pointers, then it's supposedly treated as an export ordinal value for the DLL that's being imported. It's not clear from the documentation if you really can import a function by ordinal rather than by name.

The important parts of an IMAGE_IMPORT_DESCRIPTOR are the imported DLL name and the two arrays of IMAGE_IMPORT_BY_NAME pointers. In the EXE file, the two arrays (pointed to by the Characteristics and FirstThunk fields) run parallel to each other, and are terminated by a NULL pointer entry at the end of each array. The pointers in both arrays point to an IMAGE_IMPORT_BY_NAME structure. Figure 11 shows the situation graphically. Figure 12 shows the PEDUMP output for an imports table.

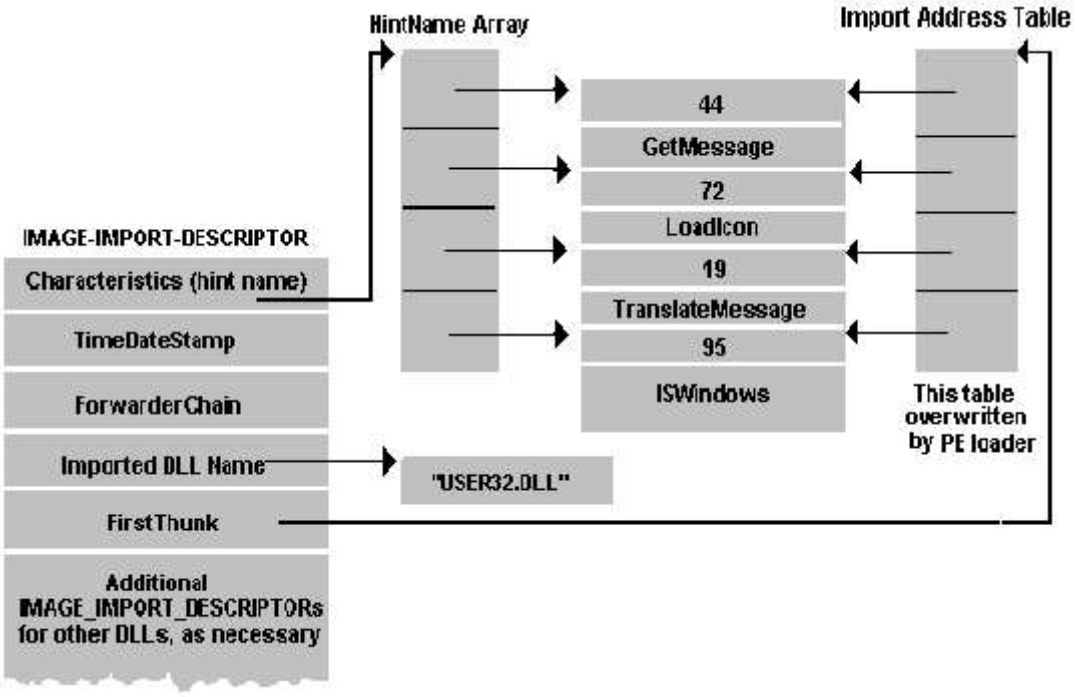

Figure 3. Two parallel arrays of pointers

Table 9. Imports Table from an EXE File

```
GDI32.dll
  Hint/Name Table: 00013064
  TimeDateStamp:   2C51B75B
  ForwarderChain:  FFFFFFFF
  First thunk RVA: 00013214
  Ordn  Name
    48  CreatePen
    57  CreateSolidBrush
    62  DeleteObject
   160  GetDeviceCaps
    //  Rest of table omitted...

  KERNEL32.dll
  Hint/Name Table: 0001309C
  TimeDateStamp:   2C4865A0
  ForwarderChain:  00000014
  First thunk RVA: 0001324C
  Ordn  Name
    83  ExitProcess
   137  GetCommandLineA
   179  GetEnvironmentStrings
   202  GetModuleHandleA
    //  Rest of table omitted...

  SHELL32.dll
  Hint/Name Table: 00013138
  TimeDateStamp:   2C41A383
  ForwarderChain:  FFFFFFFF
  First thunk RVA: 000132E8
```

```
    Ordn  Name
      46  ShellAboutA

    USER32.dll
    Hint/Name Table: 00013140
    TimeDateStamp:   2C474EDF
    ForwarderChain:  FFFFFFFF
    First thunk RVA: 000132F0
    Ordn  Name
      10  BeginPaint
      35  CharUpperA
      39  CheckDlgButton
      40  CheckMenuItem

      //  Rest of table omitted...
```

There is one IMAGE_IMPORT_BY_NAME structure for each function that the PE file imports. An IMAGE_IMPORT_BY_NAME structure is very simple, and looks like this:

```
WORD    Hint;
BYTE    Name[?];
```

The first field is the best guess as to what the export ordinal for the imported function is. Unlike with NE files, this value doesn't have to be correct. Instead, the loader uses it as a suggested starting value for its binary search for the exported function. Next is an ASCIIZ string with the name of the imported function.

Why are there two parallel arrays of pointers to the IMAGE_IMPORT_BY_NAME structures? The first array (the one pointed at by the Characteristics field) is left alone, and never modified. It's sometimes called the hint-name table. The second array (pointed at by the FirstThunk field) is overwritten by the PE loader. The loader iterates through each pointer in the array and finds the address of the function that each IMAGE_IMPORT_BY_NAME structure refers to. The loader then overwrites the pointer to IMAGE_IMPORT_BY_NAME with the found function's address. The [XXXXXXXX] portion of the JMP DWORD PTR [XXXXXXXX] thunk refers to one of the entries in the FirstThunk array. Since the array of pointers that's overwritten by the loader eventually holds the addresses of all the imported functions, it's called the Import Address Table.

For you Borland users, there's a slight twist to the above description. A PE file produced by TLINK32 is missing one of the arrays. In such an executable, the Characteristics field in the IMAGE_IMPORT_DESCRIPTOR (aka the hint-name array) is 0. Therefore, only the array that's pointed at by the FirstThunk field (the Import Address Table) is guaranteed to exist in all PE files. The story would end here, except that I ran into an interesting problem when writing PEDUMP. In the never ending search for optimizations, Microsoft "optimized" the thunk array in the system DLLs for Windows NT (KERNEL32.DLL and so on). In this optimization, the pointers in the array don't point to an IMAGE_IMPORT_BY_NAME structure—rather, they already contain the address of the imported function. In other words, the loader doesn't need to look up function addresses and overwrite the thunk array with the imported function's addresses. This causes a problem for PE dumping programs that are expecting the array to contain pointers to IMAGE_IMPORT_BY_NAME structures. You might be thinking, "But Matt, why don't you just use the hint-name table array?" That would be an ideal solution, except that the hint-name table array doesn't exist in Borland files. The PEDUMP program handles all these situations, but the code is understandably messy.

Since the import address table is in a writeable section, it's relatively easy to intercept calls that an EXE or DLL makes to another DLL. Simply patch the appropriate import address table entry to point at the desired interception function. There's no need to modify any code in either the caller or callee images. What could be easier?

It's interesting to note that in Microsoft-produced PE files, the import table is not something wholly synthesized by the linker. All the pieces necessary to call a function in another DLL reside in an import library. When you link a DLL, the library manager (LIB32.EXE or LIB.EXE) scans the OBJ files being linked and creates an import library. This import library is completely different from the import libraries used by 16-bit NE file linkers. The import library that the 32-bit LIB produces has a .text section and several .idata$ sections. The .text section in the import library contains the JMP DWORD PTR [XXXXXXXX] thunk, which has a name stored for it in the OBJ's symbol table. The name of the symbol is identical to the name of the function being exported by the DLL (for example, _Dispatch_Message@4). One of the .idata$ sections in the import library contains the DWORD that the thunk dereferences through. Another of the .idata$ sections has a space for the hint ordinal followed by the imported function's name. These two fields make up an IMAGE_IMPORT_BY_NAME structure. When you later link a PE file that uses the import library, the import library's sections are added to the list of sections from your OBJs that the linker needs to process. Since the thunk in the import library has the same name as the function being imported, the linker assumes the thunk is really the imported function, and fixes up calls to the imported function to point at the thunk. The thunk in the import library is essentially "seen" as the imported function.

Besides providing the code portion of an imported function thunk, the import library provides the pieces of the PE file's .idata section (or import table). These pieces come from the various .idata$ sections that the library manager put into the import library. In short, the linker doesn't really know the differences between imported functions and functions that appear in a different OBJ file. The linker just follows its preset rules for building and combining sections, and everything falls into place naturally.

PE File Exports

The opposite of importing a function is exporting a function for use by EXEs or other DLLs. A PE file stores information about its exported functions in the .edata section. Generally, Microsoft linker-generated PE EXE files don't export anything, so they don't have an .edata section. Borland's TLINK32 always exports at least one symbol from an EXE. Most DLLs do export functions and have an .edata section. The primary components of an .edata section (aka the export table) are tables of function names, entry point addresses, and export ordinal values. In an NE file, the equivalents of an export table are the entry table, the resident names table, and the nonresident names table. These tables are stored as part of the NE header, rather than in distinct segments or resources.

At the start of an .edata section is an IMAGE_EXPORT_DIRECTORY structure (see Table 10). This structure is immediately followed by data pointed to by fields in the structure.

Table 10. IMAGE_EXPORT_DIRECTORY Format

`DWORD Characteristics`
: This field appears to be unused and is always set to 0.

`DWORD TimeDateStamp`
: The time/date stamp indicating when this file was created.

`WORD MajorVersion`
`WORD MinorVersion`
: These fields appear to be unused and are set to 0.

`DWORD Name`
: The RVA of an ASCIIZ string with the name of this DLL.

`DWORD Base`
: The starting ordinal number for exported functions. For example, if the file exports functions with ordinal values of 10, 11, and 12, this field contains 10. To obtain the exported ordinal for a function, you need to add this value to the appropriate element of the AddressOfNameOrdinals array.

`DWORD NumberOfFunctions`
: The number of elements in the AddressOfFunctions array. This value is also the number of functions exported by this module. Theoretically, this value could be different than the NumberOfNames field (next), but actually they're always the same.

`DWORD NumberOfNames`
: The number of elements in the AddressOfNames array. This value seems always to be identical to the NumberOfFunctions field, and so is the number of exported functions.

`PDWORD *AddressOfFunctions`
: This field is an RVA and points to an array of function addresses. The function addresses are the entry points (RVAs) for each exported function in this module.

`PDWORD *AddressOfNames`
: This field is an RVA and points to an array of string pointers. The strings are the names of the exported functions in this module.

`PWORD *AddressOfNameOrdinals`
: This field is an RVA and points to an array of WORDs. The WORDs are the export ordinals of all the exported functions in this module. However, don't forget to add in the starting ordinal number specified in the Base field.

The layout of the export table is somewhat odd (see Figure 4 and Table 10). As I mentioned earlier, the requirements for exporting a function are a name, an address, and an export ordinal. You'd think that the designers of the PE format would have put all three of these items into a structure, and then have an array of these structures. Instead, each component of an exported entry is an element in an array. There are three of these arrays (AddressOfFunctions, AddressOfNames, AddressOfNameOrdinals), and they are all parallel to one another. To find all the information about the fourth function, you need to look up the fourth element in each array.

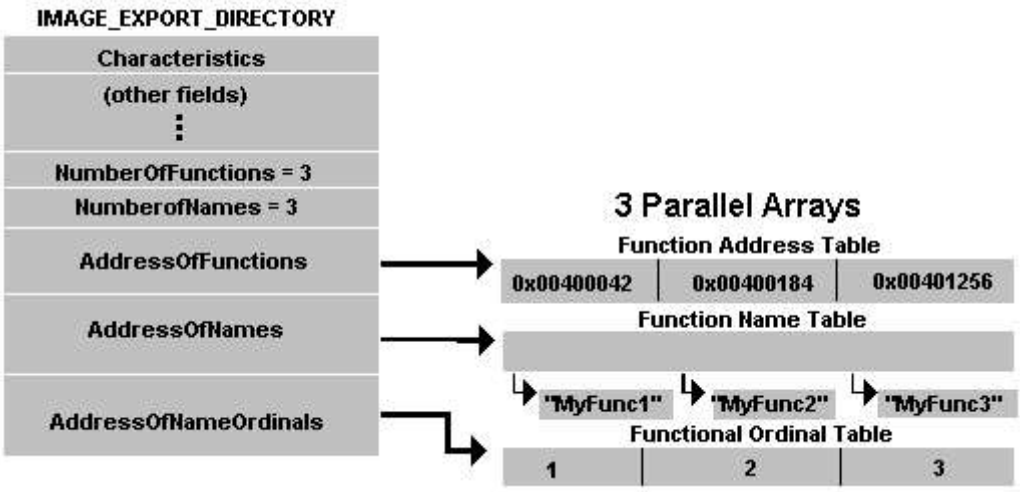

Figure 4. Export table layout

Table 11. Typical Exports Table from an EXE File

```
Name:            KERNEL32.dll
  Characteristics: 00000000
  TimeDateStamp:   2C4857D3
  Version:         0.00
  Ordinal base:    00000001
  # of functions:  0000021F
  # of Names:      0000021F

  Entry Pt  Ordn  Name
  00005090     1  AddAtomA
  00005100     2  AddAtomW
  00025540     3  AddConsoleAliasA
  00025500     4  AddConsoleAliasW
  00026AC0     5  AllocConsole
  00001000     6  BackupRead
  00001E90     7  BackupSeek
  00002100     8  BackupWrite
  0002520C     9  BaseAttachCompleteThunk
```

```
    00024C50    10  BasepDebugDump
    // Rest of table omitted...
```

Incidentally, if you dump out the exports from the Windows NT system DLLs (for example, KERNEL32.DLL and USER32.DLL), you'll note that in many cases there are two functions that only differ by one character at the end of the name, for instance CreateWindowExA and CreateWindowExW. This is how UNICODE support is implemented transparently. The functions that end with A are the ASCII (or ANSI) compatible functions, while those ending in W are the UNICODE version of the function. In your code, you don't explicitly specify which function to call. Instead, the appropriate function is selected in WINDOWS.H, via preprocessor #ifdefs. This excerpt from the Windows NT WINDOWS.H shows an example of how this works:

```
#ifdef UNICODE
#define DefWindowProc  DefWindowProcW
#else
#define DefWindowProc  DefWindowProcA
#endif // !UNICODE
```

PE File Resources

Finding resources in a PE file is quite a bit more complicated than in an NE file. The formats of the individual resources (for example, a menu) haven't changed significantly but you need to traverse a strange hierarchy to find them.

Navigating the resource directory hierarchy is like navigating a hard disk. There's a master directory (the root directory), which has subdirectories. The subdirectories have subdirectories of their own that may point to the raw resource data for things like dialog templates. In the PE format, both the root directory of the resource directory hierarchy and all of its subdirectories are structures of type IMAGE_RESOURCE_DIRECTORY (see Table 12).

Table 12. IMAGE_RESOURCE_DIRECTORY Format

`DWORD Characteristics`
: Theoretically this field could hold flags for the resource, but appears to always be 0.

`DWORD TimeDateStamp`
: The time/date stamp describing the creation time of the resource.

`WORD MajorVersion`
`WORD MinorVersion`
: Theoretically these fields would hold a version number for the resource. These field appear to always be set to 0.

`WORD NumberOfNamedEntries`

The number of array elements that use names and that follow this structure.

`WORD NumberOfIdEntries`
: The number of array elements that use integer IDs, and which follow this structure.

`IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]`
: This field isn't really part of the IMAGE_RESOURCE_DIRECTORY structure. Rather, it's an array of IMAGE_RESOURCE_DIRECTORY_ENTRY structures that immediately follow the IMAGE_RESOURCE_DIRECTORY structure. The number of elements in the array is the sum of the NumberOfNamedEntries and NumberOfIdEntries fields. The directory entry elements that have name identifiers (rather than integer IDs) come first in the array.

A directory entry can either point at a subdirectory (that is, to another IMAGE_RESOURCE_DIRECTORY), or it can point to the raw data for a resource. Generally, there are at least three directory levels before you get to the actual raw resource data. The top-

level directory (of which there's only one) is always found at the beginning of the resource section (.rsrc). The subdirectories of the top-level directory correspond to the various types of resources found in the file. For example, if a PE file includes dialogs, string tables, and menus, there will be three subdirectories: a dialog directory, a string table directory, and a menu directory. Each of these type subdirectories will in turn have ID subdirectories. There will be one ID subdirectory for each instance of a given resource type. In the above example, if there are three dialog boxes, the dialog directory will have three ID subdirectories. Each ID subdirectory will have either a string name (such as "MyDialog") or the integer ID used to identify the resource in the RC file. Figure 5 shows a resource directory hierarchy example in visual form. Table 13 shows the PEDUMP output for the resources in the Windows NT CLOCK.EXE.

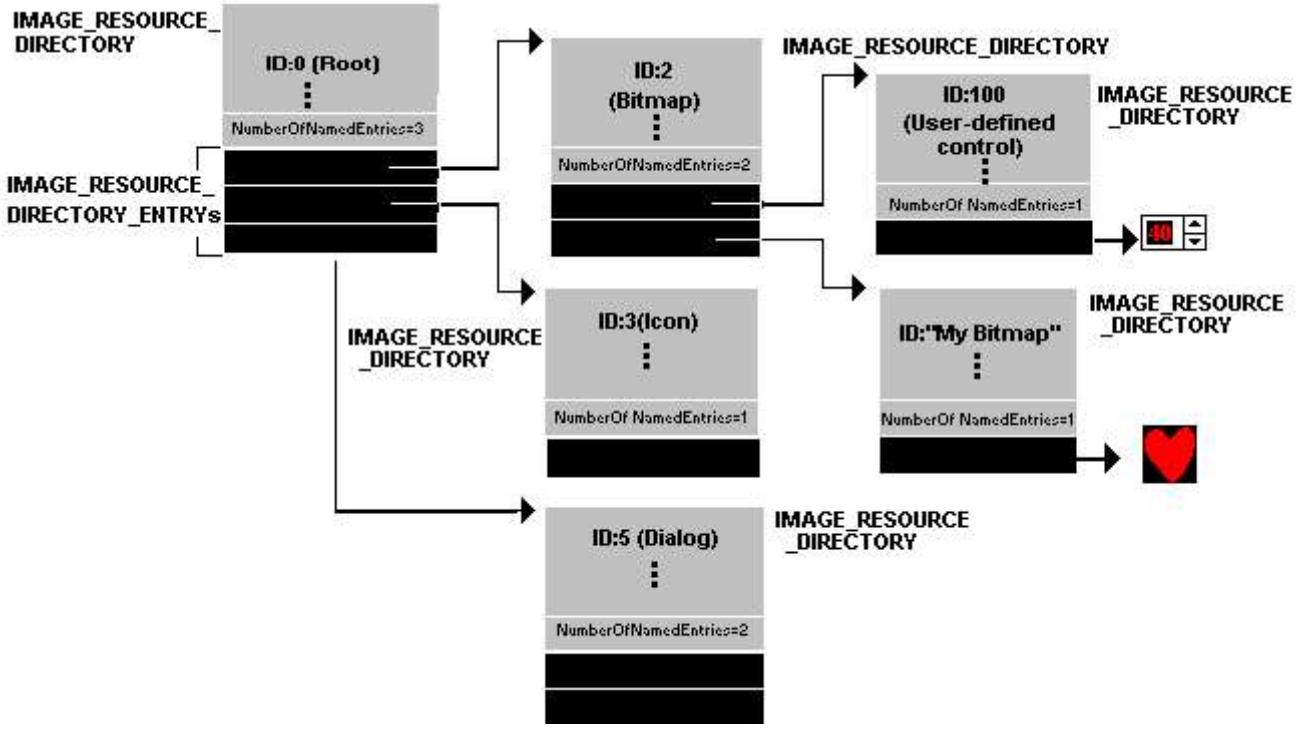

Figure 5. Resource directory hierarchy

Table 13. Resources Hierarchy for CLOCK.EXE

```
ResDir (0) Named:00 ID:06 TimeDate:2C3601DB Vers:0.00 Char:0
    ResDir (ICON) Named:00 ID:02 TimeDate:2C3601DB Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000200
        ResDir (2) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000210
    ResDir (MENU) Named:02 ID:00 TimeDate:2C3601DB Vers:0.00 Char:0
        ResDir (CLOCK) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000220
        ResDir (GENERICMENU) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000230
    ResDir (DIALOG) Named:01 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
        ResDir (ABOUTBOX) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000240
        ResDir (64) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000250
    ResDir (STRING) Named:00 ID:03 TimeDate:2C3601DB Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ID: 00000409  Offset: 00000260
```

```
            ResDir (2) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
                ID: 00000409  Offset: 00000270
            ResDir (3) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
                ID: 00000409  Offset: 00000280
        ResDir (GROUP_ICON) Named:01 ID:00 TimeDate:2C3601DB Vers:0.00 Char:0
            ResDir (CCKK) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
                ID: 00000409  Offset: 00000290
        ResDir (VERSION) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
            ResDir (1) Named:00 ID:01 TimeDate:2C3601DB Vers:0.00 Char:0
                ID: 00000409  Offset: 000002A0
```

As mentioned earlier, each directory entry is a structure of type IMAGE_RESOURCE_DIRECTORY_ENTRY (boy, these names are getting long!). Each IMAGE_RESOURCE_DIRECTORY_ENTRY has the format shown in Table 13.

Table 14. IMAGE_RESOURCE_DIRECTORY_ENTRY Format

`DWORD Name`
: This field contains either an integer ID or a pointer to a structure that contains a string name. If the high bit (0x80000000) is zero, this field is interpreted as an integer ID. If the high bit is nonzero, the lower 31 bits are an offset (relative to the start of the resources) to an IMAGE_RESOURCE_DIR_STRING_U structure. This structure contains a WORD character count, followed by a UNICODE string with the resource name. Yes, even PE files intended for non-UNICODE Win32 implementations use UNICODE here. To convert the UNICODE string to an ANSI string, use the WideCharToMultiByte function.

`DWORD OffsetToData`
: This field is either an offset to another resource directory or a pointer to information about a specific resource instance. If the high bit (0x80000000) is set, this directory entry refers to a subdirectory. The lower 31 bits are an offset (relative to the start of the resources) to another IMAGE_RESOURCE_DIRECTORY. If the high bit isn't set, the lower 31 bits point to an IMAGE_RESOURCE_DATA_ENTRY structure. The IMAGE_RESOURCE_DATA_ENTRY structure contains the location of the resource's raw data, its size, and its code page.

To go further into the resource formats, I'd need to discuss the format of each resource type (dialogs, menus, and so on). Covering these topics could easily fill up an entire article on its own.

PE File Base Relocations

When the linker creates an EXE file, it makes an assumption about where the file will be mapped into memory. Based on this, the linker puts the real addresses of code and data items into the executable file. If for whatever reason the executable ends up being loaded somewhere else in the virtual address space, the addresses the linker plugged into the image are wrong. The information stored in the .reloc section allows the PE loader to fix these addresses in the loaded image so that they're correct again. On the other hand, if the loader was able to load the file at the base address assumed by the linker, the .reloc section data isn't needed and is ignored. The entries in the .reloc section are called base relocations since their use depends on the base address of the loaded image.

Unlike relocations in the NE file format, base relocations are extremely simple. They boil down to a list of locations in the image that need a value added to them. The format of the base relocation data is somewhat quirky. The base relocation entries are packaged in a series of variable length chunks. Each chunk describes the relocations for one 4KB page in the image. Let's look at an example to see how base relocations work. An executable file is linked assuming a base address of 0x10000. At offset 0x2134 within the image is a pointer containing the address of a string. The string starts at physical address 0x14002, so the pointer contains the value 0x14002. You then load the file, but the loader decides that it needs to map the image starting at physical address 0x60000. The difference between the linker-assumed base load address and the actual load address is called the delta. In this case, the delta is 0x50000. Since the entire image is 0x50000 bytes higher in memory, so is the string (now at address 0x64002). The pointer to the string is now incorrect. The executable file contains a base relocation for the memory location where the pointer to the string resides. To resolve a base relocation, the loader adds the delta value to the original value at the base

relocation address. In this case, the loader would add 0x50000 to the original pointer value (0x14002), and store the result (0x64002) back into the pointer's memory. Since the string really is at 0x64002, everything is fine with the world.

Each chunk of base relocation data begins with an IMAGE_BASE_RELOCATION structure that looks like Table 14. Table 15 shows some base relocations as shown by PEDUMP. Note that the RVA values shown have already been displaced by the VirtualAddress in the IMAGE_BASE_RELOCATION field.

Figure 15. IMAGE_BASE_RELOCATION Format

`DWORD VirtualAddress`
: This field contains the starting RVA for this chunk of relocations. The offset of each relocation that follows is added to this value to form the actual RVA where the relocation needs to be applied.

`DWORD SizeOfBlock`
: The size of this structure plus all the WORD relocations that follow. To determine the number of relocations in this block, subtract the size of an IMAGE_BASE_RELOCATION (8 bytes) from the value of this field, and then divide by 2 (the size of a WORD). For example, if this field contains 44, there are 18 relocations that immediately follow:

```
 (44 - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(WORD) = 18
WORD TypeOffset
```

This isn't just a single WORD, but rather an array of WORDs, the number of which is calculated by the above formula. The bottom 12 bits of each WORD are a relocation offset, and need to be added to the value of the Virtual Address field from this relocation block's header. The high 4 bits of each WORD are a relocation type. For PE files that run on Intel CPUs, you'll only see two types of relocations:

0	IMAGE_REL_BASED_ABSOLUTE	This relocation is meaningless and is only used as a place holder to round relocation blocks up to a DWORD multiple size.
3	IMAGE_REL_BASED_HIGHLOW	This relocation means add both the high and low 16 bits of the delta to the DWORD specified by the calculated RVA.

Table 16. The Base Relocations from an EXE File

```
Virtual Address: 00001000  size: 0000012C
  00001032 HIGHLOW
  0000106D HIGHLOW
  000010AF HIGHLOW
  000010C5 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00002000  size: 0000009C
  000020A6 HIGHLOW
  00002110 HIGHLOW
  00002136 HIGHLOW
  00002156 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00003000  size: 00000114
  0000300A HIGHLOW
  0000301E HIGHLOW
  0000303B HIGHLOW
```

```
    0000306A HIGHLOW
    // Rest of relocations omitted...
```

Differences Between PE and COFF OBJ Files

There are two portions of the PE file that are not used by the operating system. These are the COFF symbol table and the COFF debug information. Why would anyone need COFF debug information when the much more complete CodeView information is available? If you intend to use the Windows NT system debugger (NTSD) or the Windows NT kernel debugger (KD), COFF is the only game in town. For those of you who are interested, I've included a detailed description of these parts of the PE file in the online posting that accompanies this article (available on all MSJ bulletin boards).

At many points throughout the preceding discussion, I've noted that many structures and tables are the same in both a COFF OBJ file and the PE file created from it. Both COFF OBJ and PE files have an IMAGE_FILE_HEADER at or near their beginning. This header is followed by a section table that contains information about all the sections in the file. The two formats also share the same line number and symbol table formats, although the PE file can have additional non-COFF symbol tables as well. The amount of commonality between the OBJ and PE EXE formats is evidenced by the large amount of common code in PEDUMP (see COMMON.C on any MSJ bulletin board).

This similarity between the two file formats isn't happenstance. The goal of this design is to make the linker's job as easy as possible. Theoretically, creating an EXE file from a single OBJ should be just a matter of inserting a few tables and modifying a couple of file offsets within the image. With this in mind, you can think of a COFF file as an embryonic PE file. Only a few things are missing or different, so I'll list them here.

- COFF OBJ files don't have an MS-DOS stub preceding the IMAGE_FILE_HEADER, nor is there a "PE" signature preceding the IMAGE_FILE_HEADER.
- OBJ files don't have the IMAGE_OPTIONAL_HEADER. In a PE file, this structure immediately follows the IMAGE_FILE_HEADER. Interestingly, COFF LIB files do have an IMAGE_OPTIONAL_HEADER. Space constraints prevent me from talking about LIB files here.
- OBJ files don't have base relocations. Instead, they have regular symbol-based fixups. I haven't gone into the format of the COFF OBJ file relocations because they're fairly obscure. If you want to dig into this particular area, the PointerToRelocations and NumberOfRelocations fields in the section table entries point to the relocations for each section. The relocations are an array of IMAGE_RELOCATION structures, which is defined in WINNT.H. The PEDUMP program can show OBJ file relocations if you enable the proper switch.
- The CodeView information in an OBJ file is stored in two sections (.debug$S and .debug$T). When the linker processes the OBJ files, it doesn't put these sections in the PE file. Instead, it collects all these sections and builds a single symbol table stored at the end of the file. This symbol table isn't a formal section (that is, there's no entry for it in the PE's section table).

Using PEDUMP

PEDUMP is a command-line utility for dumping PE files and COFF OBJ format files. It uses the Win32 console capabilities to eliminate the need for extensive user interface work. The syntax for PEDUMP is as follows:

```
PEDUMP [switches] filename
```

The switches can be seen by running PEDUMP with no arguments. PEDUMP uses the switches shown in Table 17. By default, none of the switches are enabled. Running PEDUMP without any of the switches provides most of the useful information without creating a huge amount of output. PEDUMP sends its output to the standard output file, so its output can be redirected to a file with an > on the command line.

Table 17. PEDUMP Switches

/A	Include everything in dump (essentially, enable all the switches)
/H	Include a hex dump of each section at the end of the dump
/L	Include line number information (both PE and COFF OBJ files)
/R	Show base relocations (PE files only)
/S	Show symbol table (both PE and COFF OBJ files)

Summary

With the advent of Win32, Microsoft made sweeping changes in the OBJ and executable file formats to save time and build on work previously done for other operating systems. A primary goal of these file formats is to enhance portability across different platforms.