What is a Linker and How Does it Work

Matt Pietrek

Note: Figures 1 and 3 at End of Article

In this column, I usually discuss technologies that are new, or at least haven't already been covered extensively. However, with more and more developers joining the ranks of Win32® programmers, topics that are old hat to veterans remain a mystery to newer programmers. The subject of linkers falls into this category. Before you Visual Basic® programmers head for the exits, be advised that Visual Basic 5.0 uses a linker. In fact, it uses the same linker that Visual C++® 5.0 does. Visual Basic 5.0 does a good job of hiding this fact, but if you snoop around you'll see that it produces OBJ files and sends them off to the Microsoft linker.

What is a linker? How does it work? This month I'll shed some light on these questions. As part of researching this column, I went back to my old sources of information. Interestingly, it seems that much of what I'll describe here is either out of print or no longer on the MSDN CD-ROM, even though linker technology affects nearly every Windows programmer.

For the purpose of this column, I'll consider Microsoft's LINK.EXE to be the standard linker. (Other linkers, such as Borland's TLINK32, may have slight differences in behavior from what I describe here.) In a future column, I'll go a step further and examine some of the more useful and interesting switches in the Microsoft linker. First, I'll give you an overly simplistic definition of a linker, and then refine it later. A linker's job is to take one or more object modules (typically OBJ files) and combine them into an executable file (that is, an EXE or DLL). However, this begs the question: what is an object module?

An object module is the output from a program that takes human-readable text and translates it into machine code and data that a CPU can understand. For C++, the C++ compiler reads a C++ source file. For assembly language, an assembler (for instance, MASM) reads an assembly language (ASM) file that contains direct equivalents for the code and data bytes that the CPU uses. In Visual Basic 5.0, the input files are the FRM, BAS, and CLS files from your project. This concept holds true for most other languages, such as Fortran.

The primary components of an object module are machine code and data. The raw bytes that make up the code and data are stored in contiguous blocks called sections. For example, Microsoft compilers put their machine code into a section called .text, and data goes into a section called .data. These names have no special meaning other than as a reminder of the intended use of a particular section. Other compilers can (and do) use different names for their sections. If you've done MS-DOS® or 16-bit Windows® programming, you can substitute the word "segment" for "sections" in the preceding description, and much of what I'll say still applies. If you have Visual C++ installed on your system, you can see sections within an OBJ file yourself with the DUMPBIN program. Execute the command

```
DUMPBIN <objname>
```

where <objname> is the name of any OBJ file. Figure 1 gives a rundown of the most commonly encountered sections. You can see the sections from a typical compile of a C++ program by running DUMPBIN on an OBJ, such as CHKSTKOBJ from the Visual C++ \LIB directory:

```
Dump of file CHKSTK.OBJ
 File Type: COFF OBJECT
      Summary
            0 .data
           2F .text
```

The fancy name for the output of a compiler or assembler is a compilation unit. However, most of us just think of them as OBJ files. The linker's most important job is to collect all the compilation units and combine all the sections from the different compilation units. Of course, if things were really this simple, the linker would be nothing more than a fancy program for concatenating blobs of data. The complicated work of a linker comes from processing fixups. More on this later.

You may be wondering how the linker decides to arrange the code and data sections from the various OBJs in the final executable. It turns out that the linker has an elaborate set of rules that must be followed. In fact, the duties of a linker are so complicated that it makes two passes through its input files. The first pass allows the linker to see what it will be working with and collect its thoughts. In the second pass, the linker applies all the rules to write out the executable file.

While I won't attempt to describe every aspect of every linker rule here, there are a couple of rules that cover the majority of linker behavior. The primary rule is that the linker must put all of the code and data from every specified OBJ file into the executable. If you give the linker three OBJ files, then the code and data from all three OBJ files must somehow be incorporated into the executable. However, the linker doesn't simply take the raw sections from each OBJ file and string them end to end in the executable. Rather, the linker combines (concatenates) all sections with the same name. For example, if the three OBJ files each had a .text section, the resulting executable will have a single .text section, comprised of the three individual .text sections concatenated together in the order in which they were encountered.

Another rule observed by the linker is that the sequence of sections in the executable file is dictated by the order in which the linker processes the sections. The linker works its way through the list of OBJ files in exactly the order specified on the command line. However, the primary rule of combining sections with the same name takes precedence.

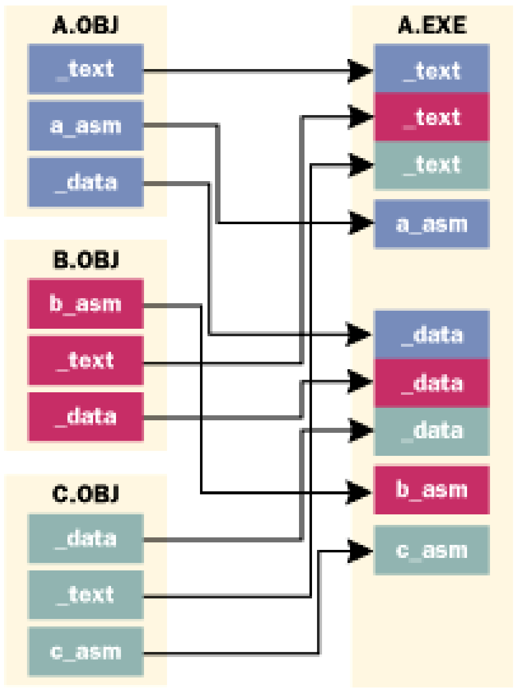

Figure 2 A.OBJ, B.OBJ, and C.OBJ

Figure 2 shows three OBJ files, A.OBJ, B.OBJ, and C.OBJ. In each file are three sections—all have _text and _data sections, but in different positions within each OBJ. They also all have a third section unique to their source file (that is, a_asm, b_asm, and c_asm). Let's say you invoked LINK, passing it the command line

```
OBJ B.OBJ C.OBJ
```

The order of segments (and how sections with the same name are combined) is shown in the right-hand side of Figure 2. You can download the source and OBJ files from the link at the top of this article. This way, you can experiment with different linker command lines—for example "Link B.OBJ A.OBJ C.OBJ"—even if you don't have MASM or a compatible assembler.

With these two rules in mind, you're a good way toward knowing how the linker does its job under MS-DOS and 16-bit Windows. The Win32 linker adds several twists to what I just described, though. For starters, there's the $ section name rule. If a section name contains a $ in it (for example, .idata$4), the $ and everything that follows will be stripped off in the executable file. However, before the linker strips down the name, it combines the sections with names that match up to the $. The name portion after the '$' is used in arranging the OBJ sections in the executable. These sections are sorted alphabetically, based on the portion of the name after the $. For example, three sections called fooc, fooa, and foo$b will be combined into a single section called foo in the executable. The data in this section will start with foo$a's data, continue with foo$b's, and end with foo$c's. This automatic combining of sections with $ in their name is used in a variety of ways. You'll see one example later when I discuss imported functions. It's also used to create the data tables needed for static initialization of C++ constructors and destructors.

Besides the $ combination rule, the Win32 linker has a few other special cases up its sleeve. Sections with the code attribute are given special preference and put first in the executable file. Following any code, the linker puts any uninitialized data sections—comprised of global data for which an initial value wasn't specified at the time of compilation (for instance, int i; declared as a global variable in C++). Next comes initialized data (including the .data section), as well as linker-generated data sections such as .reloc.

Uninitialized data is usually put into a section called .bss by a compiler. It's rare to see a .bss section in an executable file these days, though. The Microsoft linker merges the .bss section into .data, which is the main initialized data section used by compilers. But wait, there's another catch! This only happens if the executable is for a subsystem other than Posix, and the subsystem version is greater than 3.5. Other sections that contain uninitialized data are left alone by the linker (that is, they aren't merged).

Working backwards from the end of the executable, if there is a .debug section in the OBJs, it's placed last in the executable. In the absence of a .debug section, the linker tries to put the .reloc section last because, in most cases, the Win32 loader won't need to read the relocation information. Cutting down the amount of the executable that needs to be read decreases the load time. I'll describe relocations later.

Yet another exception to the basic two rules that comes up under Win32 is removable sections. These sections exist in an OBJ file, but the linker doesn't copy them into the executable. These sections typically have the LINK_ REMOVE and LINK_INFO attributes (see WINNT.H), and are named .drectve. Microsoft compilers spit them out to pass on information to the linker. If you look at an OBJ that was compiled with Visual C++, you'll see that the data in the .drectve section probably looks something like this:

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

If this data looks suspiciously like command-line arguments to the linker, you're on the right track. You can see other evidence of this when you use the __declspec(dllexport) modifier with C++. For example:

```
void __declspec(dllexport) ExportMe( void ){...}
```

will cause the .drectve section to also contain:

```
-export:_ExportMe
```

Sure enough, if you look at the list of command-line options to LINK, -export is one of them.

Fixups and Relocations

Why can't compilers simply generate executable files directly from the source file, thereby eliminating the need for a linker? The primary reason is that most programs don't consist of a single source file. Compilers specialize in taking a single source file and producing a raw, machine-code equivalent. Because a source file may contain references to code or data external to the source file, a compiler can't generate exactly the right code to call that function or access that variable. Instead, the compiler's only option is to include extra information in the output file that describes the external code or data. The term for this description of external code and data references is a fixup. Putting it bluntly, the code that the compiler generates for accessing external functions and variables is incorrect, and must be fixed up later.

Consider a call to a function named Foo in C++:

```
//...
 Foo();
 //...
```

The exact bytes emitted from a 32-bit C++ compiler will be this:

```
E8 00 00 00 00
```

The 0xE8 is the CALL instruction opcode. The next DWORD should contain the offset to the Foo function (relative to the CALL instruction). It's pretty clear that Foo probably isn't zero bytes away from the CALL instruction. Simply put, this code wouldn't work as expected if you were to execute it. The code is broken, and needs to be fixed up.

In the above example, the linker needs to replace the DWORD following the CALL opcode with the correct address of Foo. In the executable file, the linker will write a DWORD with the relative address of Foo. How does the linker know this needs to be done? A fixup record tells it so. How does the linker know where function Foo is? The linker knows about all symbols in an executable because it's responsible for arranging and combining the components of the executable.

Now, about those fixup records. For Intel-based OBJ files, there are only three types of fixup records that are normally encountered. The first are 32-bit relative fixups, known as REL32 fixups. (For the curious, they correspond to the IMAGE_REL_I386_REL32 #define in WINNT.H.) In the above example of a call to function Foo, there will be a REL32 fixup record, and it will have the offset of the DWORD that the linker needs to overwrite with the appropriate value. If you were to run

```
DUMPBIN /RELOCATIONS
```

for the OBJ created by the above code, you'd see something like this:

```
Symbol    Symbol
  Offset     Type     Applied To  Index     Name
  ________   ____     __________  ______    ______

00000004  REL32    00000000      7       _Foo
```

In English, this fixup record says that the linker needs to calculate the relative offset to function Foo, and write that value to offset four in the section. Since this fixup record is only needed by the linker prior to creating the executable, it's discarded and doesn't appear in the executable. Why then do most executables contain a section called .reloc? This is where the second type of fixup comes into play. Consider the following program:

```
int i;
 int main()
 {
     i = 0x12345678;
 }
```

Visual C++ generates this instruction for the assignment in the executable:

```
MOV DWORD PTR [00406280],12345678
```

What's interesting is the [00406280] part of the instruction. It's referencing a fixed location in memory, and assumes that the DWORD containing the variable i is 0x6280 bytes above the load address of the executable file, which is at 0x400000 by default. Now, consider what happens if the executable can't be loaded at the default load address. Instead, let's say that the Win32 loader loads it 2MB higher in memory (that is, the executable loads at 0x600000). When this happens, the [00406280] part of the instruction needs to be adjusted to 0x00606280.

It's for just such occasions that DIR32 (Direct 32) fixups are used in OBJ files. They signify locations where the actual (direct) address of something needs to be plugged in. By implication, they also signify the locations where the load address of the executable file is significant. When creating the executable, the loader takes the DIR32 fixups from the OBJs and creates the .reloc section. Before this happens though, running DUMPBIN /RELOCATIONS on the OBJ shows:

```
Symbol   Symbol
  Offset     Type     Applied To  Index    Name
  ________   _____    __________  ______   ______

00000005  DIR32    00000000      4      _i
```

The fixup record says that the linker needs to calculate the direct 32-bit address of the variable _i, and write that value to offset five in the section.

The .reloc section in an executable is basically a series of addresses in the executable where the difference between the default and actual load address needs to be accounted for. By default, the linker creates the executable so that the .reloc section isn't needed by the Win32 loader. However, when the Win32 loader needs to load an executable somewhere other than its preferred load address, the .reloc section allows all the direct references to code and data to be updated.

The third type of fixups commonly found in Intel OBJs, DIR32NB (Direct 32, No Base), are used for debug information. One of the secondary jobs of the linker is to create debug information that includes the names of functions and variables, along with their addresses. Since only the linker knows where all the functions and variables will end up, the DIR32NB fixup is used to indicate spots in the debug information where the address of a function or variable is needed. The key difference between DIR32 and DIR32NB fixups is that the values patched in for DIR32NB fixups don't include the default load address of the executable.

Libraries

In some circumstances, it's worthwhile to combine two or more OBJs together into a single file, which can then be given to the linker. The classic example of this is the C++ runtime library (RTL). The C++ RTL is made up of numerous source files that are compiled, and the resulting OBJs are combined into a library. For Visual C++, the standard, single threaded, static version of the runtime library is called LIBC.LIB. There are other variations for debugging (for example, LIBCD.LIB) and multithreading (LIBCMT.LIB).

Library files usually have the .LIB extension. They consist of a library header, followed by the raw data of the contained OBJs. The library header informs the linker which symbols (functions and variables) can be found in the following OBJs, as well as which OBJ a given symbol resides in. You can see the contents of a library via the DUMPBIN /LINKERMEMBER switch. Without going into the details of why, you'll find DUMPBIN's output more readable if you specify :1 or :2 afterwards. For example, using PENTER.LIB from Visual C++ 5.0 with the command

```
"DUMPBIN /LINKERMEMBER:1 PENTER.LIB"
```

produces this snippet of output:

```
6 public symbols
       180 _DumpCAP@0
       180 _StartCAP@0
       180 _StopCAP@0
       180 _VERSION
       180 __mcount
       180 __penter
```

The 180 in front of each symbol name indicates that the symbol (for instance, _DumpCAP@0) can be found in an OBJ file beginning 0x180 bytes into the library. As you can see, PENTER.LIB only has one OBJ in it. More complicated LIB files will have multiple OBJs, so the offsets preceding the symbol names will be different.

Unlike OBJs passed on the command line, the linker does not have to include every OBJ in a library into the final executable. Quite the opposite, in fact. The linker won't include any OBJ code or data from a library OBJ unless there's a reference to at least one symbol from that OBJ. Put another way, explicitly named OBJs on the linker command line fly first class, and are always included in the executable. OBJs from LIB files fly standby, and are only included in the executable if referenced.

A symbol in a library can be referenced (and hence, its OBJ included) in three ways. First, there can be a direct reference to a symbol from one of the explicit OBJ files. For example, if I were to call the C++ printf function from a source file I wrote, there would be a reference (and a fixup) generated for it in my OBJ file. When creating the executable, the linker would search its LIB files for the OBJ containing the printf code, and include the OBJ it finds.

Second, there can be an indirect reference. Indirect means an OBJ included via the first method contains references to symbols in yet another OBJ file in the library. This second OBJ may in turn reference symbols in a third OBJ file in the library. One of the linker's toughest jobs is to track down and include every OBJ that has a referenced symbol, even if that symbol is located via 49 levels of indirection.

When looking for a symbol, the linker searches the LIB files in the order it encountered them on the command line. However, once a symbol is found in a library, that library becomes the preferred library, and is given first crack at all future symbols. The library loses its favored status once a symbol isn't found in the library. In this case, the next library in the linker list is searched. (For a more technically detailed description, see the Microsoft Knowledge Base article Q31998.)

Let's now address the issue of import libraries. Structurally, import libraries are no different than regular libraries. When resolving symbols, the linker doesn't know the difference between an import library and a regular library. The key difference is that there's no compilation unit (for example, source file) that corresponds to each OBJ in the import library. Instead, the linker itself produces the import library, based upon the symbols that are exported from an executable being built. Put another way, when the linker creates the exports table in an executable, it also creates the corresponding import library to reference those symbols. This point leads nicely to my next topic, the imports table.

Creating the Imports Table

One of the most fundamental features that Win32 rests upon is the ability to import functions from other executables. All of the information about the imported DLLs and functions resides in a table in the executable known as the imports table. When it's in a section all by itself, this section is named .idata.

Since imports are so vital to Win32 executables, it may seem strange that the linker doesn't have any special knowledge of import tables. Put another way, the linker doesn't know or care whether a function you've called resides in another DLL, or within the same executable. The way that this is accomplished is all very clever. By simply following the section combining and symbol resolution rules described above, the linker creates the imports table, seemingly unaware of the special significance of the table.

Let's look at some fragments from an import library to see how the linker accomplishes this feat. Figure 2 shows portions of running DUMPBIN on the USER32.LIB import library. Pretend that you've called ActivateKeyboardLayout API. A fixup record for _ActivateKeyboardLayout@8 can be found in your OBJ file. From the USER32.LIB header, the linker determines that this function can be found in the OBJ at offset 0xEA14 in the file. At this point, the linker is committed to including the contents of this OBJ in the finished executable (see Figure 3).

From Figure 3, you can see that a variety of sections from the OBJ will be brought in, including .text, .idata$5, .idata$4, and .idata$6. In the text section is a JMP instruction (the 0xFF 0x25 opcode). From the COFF symbol table at the end of Figure 3, you can see that _ActivateKeyboardLayout@8 resolves to this JMP instruction in the .text section. Thus, the linker hooks up your CALL to ActivateKeyboardLayout to the JMP instruction in the .text section of the import library's OBJ.

The linker combines the .idata$XXX sections into a single .idata section in the executable. Now recall that the linker has to follow the rule for combining sections with a $ in their name. If there are other imported functions brought in from USER32.LIB, their .idata$4, .idata$5 and .idata$6 sections will also be thrown into the mix. The net result is that all the .idata$4 sections create one array, while all the .idata$5 sections create another array. If you're familiar with the term "import address table," this process is how that table is created.

Finally, notice that the raw data for the .idata$6 section contains the string ActivateKeyboardLayout. This is how the name of imported functions make it into the import address table. The important point is that creating the import table isn't a big deal for the linker. It's just doing its job, following the rules I described earlier.

Creating the Exports Table

Besides creating an import table for executables, a linker is also responsible for creating the opposite: the exports table. Here, the linker's job is both harder and easier. In pass one, the linker has the task of collecting information about all the exported symbols and creating an exported function table. During the first pass, the linker creates the export table and writes it to a section called .edata in an OBJ file. This OBJ file is standard in all respects, except that it uses an extension of .EXP rather than .OBJ. That's right, you can use DUMPBIN to examine the contents of those EXP files that seem to accumulate in the presence of DLLs that you build.

During its second pass, the linker's job is almost trivial. It simply treats the EXP as a regular OBJ file. This in turn means that the .edata in the OBJ will be included in the executable. Sure enough, if you see an .edata section in an executable, it's the export table. These days, though, finding an .edata section is increasingly rare. It seems that if the executable uses the Win32 console or GUI subsystems, the linker automatically merges the .edata section with the .rdata section, if one is present.

Wrap Up

Obviously, a linker has many more jobs than I've described here. For example, producing certain types of debug information (such as CodeView info) is a major piece of a linker's total work. However, creating debug information isn't an absolutely mandatory job for the linker, so I haven't spent any time describing it. Likewise, a linker should be able to create a MAP file listing the public symbols that were included in the executable, but again it's not a mandatory function of a linker.
While I've covered a lot of complex ground, at its heart a linker is simply a tool for combining multiple compilation units into a functioning executable. The first cornerstone is in combining sections; the second is in resolving references (fixups) between the combined sections. Throw in a dash of knowledge about system-specific data structures such as the exports table, and you've covered the basics of this powerful and essential tool.

Figure 1 Commonly Encountered Sections

Figure 1 Commonly Encountered Sections	
.text	Machine code instructions.
.data	Initialized data.
.rdata	Read only data. OLE GUIDs are stored here, among other things.
.rsrc	Resources. Produced by the resource compiler, and placed into

	RES files. Linker copies it to the executable.
.reloc	Base relocations. Produced by the linker. Not found in OBJs.
.edata	The exported function table. Created by the linker and placed in an EXP file. Linker copies it to the executable.
.idata	Imported function table in an executable file.
.idata$XXX	Portions of an imported function table. The librarian creates these sections in an import library. The linker combines them into the final .idata section in the executable.
.CRT	Tables of initialization and shutdown pointers in the executable that are used by the Microsoft C++ runtime library.
.CRT$XXX	Initialization and shutdown pointers in OBJs, prior to the linker combining them in the executable.
.bss	Uninitialized data.
.drectve	OBJ file section containing linker directives. Not copied to executable.
.debug$XXX	COFF symbol table information in an OBJ file.

Figure 3 An Imports Table

```
1121 public symbols

     EA14 _ActivateKeyboardLayout@8
...
 Archive member name at EA14: USER32.dll/
...

SECTION HEADER #2
   .text name
RAW DATA #2
00000000  FF 25 00 00 00 00                                 .%....

...

SECTION HEADER #4
.idata$5 name
RAW DATA #4
00000000  00 00 00 00                                       ....

...

SECTION HEADER #5
.idata$4 name
RAW DATA #5
00000000  00 00 00 00                                       ....

...

SECTION HEADER #6
.idata$6 name
RAW DATA #6
00000000  00 00 41 63 74 69 76 61 | 74 65 4B 65 79 62 6F 61   ..Activa|teKeyboa
00000010  72 64 4C 61 79 6F 75 74 | 00 00                     rdLayout|..

...
COFF SYMBOL TABLE
...
003 00000000 SECT2  notype ()    External     | _ActivateKeyboardLayout@8
```