

## OS/2 2.1 Executable Files

Executable files today are the end result of a massive collaboration of make files, source files, include files, compiler flags, linker flags, environment variables, definition files, and resource files, as well as source control flags, if you're into that. Get something wrong in one of them, and even the prettiest algorithm turns into a UAE.

Any Windows programmer who's ever forgotten to export a dialog window procedure knows exactly what I'm talking about. The dialog comes up and sort of runs, then crashes. You stare at the code and it looks great. Your good friend, the cleaning lady, stops by to empty your trash. The soda machine runs out of Jolt. Your patience evaporates. Or ... your multi-threaded program seems to croak in printf. You know you have to link with the DLL version of the run-time lib (libcdll.dll), but you have this nagging suspicion you might still be catching the static run-time lib ...

What you need is another tool, one that can look into the executable and tell you exactly what's going on inside, in the first instance, dumping the **Resident** and **Non-Resident Name Tables** where the offending window procedure should appear, and in the second, dumping the **Imported Name Table** where llibcdll.dll should appear.

Out of such desperate situations was SHOWEXE.C born. SHOWEXE was written in the bad old, pre-beta OS/2 0.X and Windows 1.0 days when the NE style segmented executable format was the latest and greatest. Adapted from a **PC Tech Journal** article by David Schmitt, it originally exploded NE style executables. I recently updated it to do the same for the new, 32-bit, flat memory-model LX style executables. In the process, I looked through a number of real executable files, as well as some designed just for this purpose. Here's what I found.

### Headers

An NE or LX exe file will always contain at least two parts: the old DOS 2.1 MZ exe and the new NE or LX exe. The first part of the file is the MZ exe (so-called for the two ASCII bytes at offset 0 in the file) which contains the DOS stub program that prints the message "*This program requires Microsoft Windows*". The MZ header contains a pointer to the NE or LX header (**lfanew** in Listing 1) that is the file offset of the new style exe header. Listing 1 shows a simplified MZ header structure that gets you the new exe file offset. Between the MZ header and file offset **lfanew** there may reside an actual DOS program of variable size.

The two ASCII *chars* at **lfanew** contain the executable format specifier: NE, LE, LX (or PE for Win NT). NE indicates a 16-bit, segmented Windows or OS/2 exe, LX a 32-bit, flat memory model OS/2 2.1 exe. Windows 3.1 is made up almost entirely of NE executables while OS/2 2.1 contains a mix of NE and LX executables. Table 1 shows the file types of some of the files delivered with Windows 3.1 and OS/2 2.1. SMARTDRV and EMM386 are the only LE executables I was able to find in either of these products.

### Crossing the Great Processor Divide

A quick look at the header flags shows that there are several new moves in the direction of cross-processor portability in LX. The most significant are the **Byte Order** and **Word Order** specifiers. Occurring directly after the **Executable Type** specifier (to minimize the amount of wrong-endian processing any loader might have to do) these now allow either big or little-endian byte and word orders over the entire rest of the executable file. **Processor Type** and **OS Type** specifiers also now each get 16 bits of their own, where NE relegated them to a couple of bits each in the Flags word. Interestingly, the **Memory Page Size**, which is fixed at 4k on Intel X86 CPUs, is also parameterized in the header, presumably to vary over different hardware platforms.

## Virus Protection

NE and LX take different approaches to preserving executable integrity. NE allowed space for a 32-bit file checksum in the header. This was intended as a layer of protection against viruses, to be checked off-line via a separate virus-checker. LX takes a finer-grained, load-time approach to executable integrity. There are individual checksums for each 4k page as well as the **Fixup Section**, the **Non-Resident Name table**, and the **Loader Section**, which includes all tables except the **Non-Resident Name table**. In sum, these cover the entire file, but individually they allow the loader to do much less expensive checksums against small pieces of the file as they're loaded, rather than doing the whole file at once as the NE checksum requires.

## Initial Values

Initial values in both models are pretty much what you'd expect: code segment/offset, data segment/offset, stack size, heap size ... They change from 16-bit values in NE to 32 bit values in LX, but their intent remains the same.

## Segments, Pages and Objects

Now for the good stuff. The real difference between OS/2 1.X and OS/2 2.X is the shift from the 16-bit segmented memory model to the 32-bit, flat memory model. This model shift is best reflected in the NE **Segment Table** and it's LX analogs, the **Object** and **Object Page Tables**. To see how it works in practice, consider the program in Listing 2, which is simply two global arrays of 32k integers, with a single reference to each array.

When compiled and linked as an NE exe using *Microsoft C6*, the linker produces one code **Segment** from our minuscule program, and three data **Segments**; a distinct **Segment** for each array (**Segments** 3 and 4) and one for the **Auto-DS Segment** (segment 5) as shown in Table 10. The same program linked 32-bit under *Borland C++ for OS/2* produces the list of **Objects** and **Pages** of Table 11. As with NE, we get only one code **Object** which has one page, but the two large arrays produce only one data **Object** made up of 64 zero-filled, 4k pages (32767 ints \* 4 bytes per int ->262,136 bytes/4096 bytes/page = 64 pages). Thus, LX replaces the NE **Segment Table** with the **Object** and **Object-Page Tables**.

An NE **Segment Table** entry contains a set of flags (READ/WRITE/EXECUTE ...), the file location of the Segment image, the file size of the segment and it's size in memory. An LX **Object Table** entry looks much the same, containing the object's size and attributes (read/write/execute...) but also containing a count of pages that make up the object and a pointer into the **Object Page Table** where this object's first page is described (and the rest are stored consecutively). Listing 1 shows the structures of NE's **Segment Table**, and LX's **Object** and **Object Page Table** entries.

## Relocation and Internal Fixups

In order for these Objects/Segments to make a coherent program, the code object/segment has to be able to access addresses in the other objects/segments. Relocation is the process of connecting references within a lump of executable code to things outside that lump. In general, relocation consists of three items of data, the **target**, **source** and **Fixup Record**. The **target** is the place in the code where a symbolic reference must be replaced by a real address (the target of the relocation process), the **source** is the place where the real address can be found, and the **Fixup Record** is what links the target to the source. The **Fixup Record Table** is a list of **Fixup Records** for a particular segment/object.

There are two basic types of relocations; internal and external. External fixups are references that resolve to something outside this executable, typically DLL calls. Internal fixups are references to objects within this executable, typically references to the data segment. For the purpose of discussing 32-bit vs. 16-bit exes, we'll look at internal fixups first.

NE and LX structure the relocation data fundamentally differently. NE attaches a separate **Fixup Record Table** to each segment that needs one. If a segment contains **targets**, NE physically appends a **Fixup Record Table** to the end of that segment and sets the Relocations Available bit in the segment description in the **Segment Table**. The **Fixup Records** themselves contain the offset of the **target** and Segment # and offset of the **source**.

LX linkers place a single **Fixup Record Table** in the header. Each page in the **Object-Page Table** contains an index into the **Fixup Record Table** pointing to the first **Fixup Record** for that page. The linker stores all the **Fixup Records** for that page consecutively, right up to the first **Fixup Record** for the next page. The **Fixup Record** contains the offset of the **target** and the **Object Number**, and offset of the **source**. The **target** itself contains only the offset of the **source**. When loading a particular page, the loader runs through the **Fixup Record Table**, reading the attributes of the **target** and **source**, the **source** object number and the location of the **target** from the **Fixup Record** and then getting the actual offset of the **source** within the **source object** from the contents of the **target**.

While their **Fixup Records** are roughly equivalent, NE requires one record for each **source**, while LX uses the more obvious one-per-**target** strategy. To illustrate, in Listing 6, we have three large arrays and two references to each array. As shown in Tables 6 and 7, NE produces three **Fixup Records** while LX produces six. Obviously, the NE strategy is a little more load-time efficient, while LX's is more run-time efficient. NE's strategy also allows for chaining of **targets**, which we discuss with External Fixups.

## External Fixups and Imported Names

External fixups refer to objects that are located physically outside this executable, the most common being DLL calls. Both NE and LX executables support import-by-ordinal and import-by-name external fixups. The linker generates import-by-name external fixups for called functions that are defined explicitly in the **IMPORTS** section of the .DEF file. It generates import-by-ordinal external fixups for called functions that are defined in import libraries, a much more common technique. Import-by-name enjoys the benefit that you don't need the import library to link the exe, but in my experience this technique is VERY rare.

With import-by-name DLL calls, the linker stores both the name of the external DLL, and the name of the function within the DLL in the exe itself. In LX, the **target** field in the **Fixup Record** contains indexes into the **Import Module Name Table** and the **Import Procedure Name Table** which gives you the name of both the DLL and the function within the DLL, while in NE, the **target** field contains two indexes into the **Imported Name Table** that do the same job.

With an import-by-ordinal reference, the linker specifies the DLL name and the function's ordinal within that DLL. The **Fixup Record** target field contains an index into the **Import Module Name Table (Imported Name Table** in NE), but instead of an index into the **Import Procedure Name Table** to get the function name, it just contains the function's ordinal number within the external DLL. Thus, in order to get the function name to go with the ordinal, you have to run something like SHOWEXE on the external DLL and find the ordinal in either the **Resident** or **Non-Resident Name Tables**.

While the Fixup Records and sources for external fixups are much the same in NE and LX, the targets can be very different. Within NE, targets within the same segment that resolve to the same source are chained together, so that the loader has only to read one **Fixup Record**. In Listing 7 we have a program that simply makes three calls to DosSleep(). You might expect this to generate three Fixup Records, but as you can see in Table 8, we only get one. In Listing 8 we disassemble Listing 7 and see that the contents of the three target calls to DosSleep are 0000:001B, 0000:0024, and 0000:FFFF. There's our chain, the **Fixup Record** points to the target at offset 0012, which points to the target at offset 001B to the target at offset 0024, which signals the end of the chain with FFFF. At load-time, the loader gets a real address for the **source**, then marches up the **target** chain replacing the chain links with the real address. (You have to look at the disk file to see the target chain. If you CodeView it, CodeView will replace the target chain links with a real address.) In LX, there is no support for target chaining. The **Fixup Record Table** for the LX version of Listing 7 is shown in Table 9. As you can see, the linker produces a separate **Fixup Record** for each call to DosSleep().

## Other Tables

OS/2 2.X retains many basic concepts of OS/2 1.X, such as resource-handling and dynamic-linking. Thus, the **LX Module Reference**, **Entry**, **Resident** and **Non-resident Name**, and **Resource** tables are largely the same as in NE. The pointers to these tables, their byte counts, and their contents are, of course, all 32-bit in LX. The single **Imported Name Table** from NE becomes the **Imported Module** and **Imported Procedure Tables** in LX, though the purpose, structure and functionality remain the same. The **Entry Table**, of course, allows other executables to reference this exe via an External Fixup so the same issues, and changes that were required for **Fixup Records** also apply generally to **Entry Table** entries.

## Reading the Tables

In either format, the linker places the "file offsets" of all the tables (except the **Fixup Record Table** in NE) in the header, but you have to be careful with these offsets. Though most of them are relative to the beginning of the new header, some are relative to other offsets or to the beginning of the file. Some of the tables have corresponding element counts within the header, but some are terminated by a special character, and others are only terminated by reaching the file offset of the next file section. And just to make it interesting, some of the tables such as the **Entry** and **Fixup Record Tables** contain entries which are structures made up of members of variable (8, 16 and 32 bit) size. Tables 4 and 5 contain lists of the tables, where they're located, what type of elements they contain, and how they're terminated.

The structures of the fixed-size entries can be seen in Listing 1. To read them, just keep reading the fixed-size struct until you hit the terminator. The only table you have to dig for is the NE **Fixup Record Table**. As we talked about earlier, the NE linker appends **Fixup Record Tables** to the segments to which they belong. Loading them requires finding the segment through the offset in the **Segment Table** record, adding the segment size, reading the two-byte relocation count at that offset, and then reading that many **Fixup Records**.

The variable-sized entries require a little more explanation. Under NE, the "Name" table entries (**Imported**, **Resident** and **Non-Resident**) all contain a Pascal-style string (1 byte *length* followed by *length* bytes, non-null-terminated string). **Resident** and **Non-Resident Name Table** entries also add a two-byte ordinal to the end. Under LX, the "Name" table entry structures (**Import Module**, **Import Procedure**, **Resident** and **Non-Resident**) are identical except that they expand the string-length to two bytes.

The **Entry Table** in NE and the **Entry** and **Fixup Record Tables** in LX contain more complex variable-sized entries. All three of these tables have to be read as byte streams. **Entry Table** entries are bundled, with the first two bytes being the count of entries in the bundle and the type of those entries, followed by count entries of a single format. So, you hit the bundle, figure out the count and the encapsulated format, and read *count* encapsulated entries. Easy enough, eh? Figures 4 and 5 show the format of **Entry Table** entries for NE and LX.

Unlike NE, the LX **Fixup Record Table** is part of the file header. It is probably the most painful LX table to take apart. The **Fixup Records** themselves can take one of four formats, and within each of these formats, two or three of the fields have a variable size. The **Fixup Record** types you'll run into most often are 16-bit Internal Fixups and Import-by-Ordinal External Fixups. Figures 6 and 7 show the format of **Fixup Records** for NE and LX.

## Physical Objects

Having dealt with the headers and the tables, we find only two more types of data in the exe; Debugging Info and Segments/Pages. Both formats punt on Debugging Info, allowing the linker to reserve a section of the executable for proprietary-format debugging data. LX does make a weak attempt to assist the debugging process, instituting a pointer to a linker/debugger-specific **Debugging Info Section** and **Debugging Info Length**.

In NE, the linker places the file offsets of all the code and data segments in the **Segment Table**. The actual segments are located on sector boundaries and are found by shifting the sector index in the **Segment Table** entry (*ns\_sector* in the SEG struct) left by the alignment (*align* from NE struct). Get the offset and read *ns\_cbseg* bytes and you have the actual segment.

In LX the linker puts the offset of the page within the **Data** or **Iterated Page Section** in the page's **Object Page Table** entry, while placing the sizes and file offsets of the actual **Data** and **Iterated Page Sections** in the header. You locate the physical pages within the file by reading the page data offset from the **Object Page Table**, shifting it left by the **Page Offset Shift** and adding it to either the **Data** or **Iterated Data Pages Offset** (depending on **flags** in the OBJPG struct). All pages are physically 4k bytes long with any sub-4k pages zero-filled to reach the required size. Remember that both formats support zero-filled pages/segments that exist as entries in the **Object Page** or **Segment Tables** but don't have physical images in the executable.

## Documentation

I never saw any documentation on the NE format though I have heard rumors that there is some available now from Microsoft. I relied on David Schmitt's excellent PC Tech Journal article, my trusty debugger and lots of experimentation. LX is actually pretty well documented by IBM in a document titled *IBM OS/2 32 bit Object Module Format and Linear Executable Module Format* which is also available on Compuserve, GO OS2SUP, library 17, file *OMF.ZIP*.

## Onward and Upward

With the information we now have in hand there are a number of interesting projects we could undertake. The most obvious is a disassembler. We've located the code lumps, and fixed up the fixups so the rest, as they say, is a simple matter of programming. A much easier project is to reverse-engineer the resource file. We've located the resource bundles, and the resource attributes are copied pretty much one-to-one to the bundle entries, so it's not all that difficult.

In any case, the days when the loader simply copied all the bytes into memory and jumped to the entry point are long gone. With the advent of OOP, GUIs, internationalization, and the never-ending drive toward portability, exe file formats have become more and more interesting. The loader and operating system now encompass functionality that would previously have been written into the source, if it were written at all. To master that functionality we need to understand the reaction of the loader and the OS to particular facets of the executable, and be able to see clearly all the parts of that executable.

*Special thanks to Mike Roth at IBM for his help with this article.*

Format	Name	Description
NE	SOL.EXE	Solitaire game.
LE	EMM386.EXE	Extended memory manager.
NE	PRINTMAN.EXE	Print manager.
NE	PROGMAN.EXE	Program manager shell.
LE	SMARTDRV.EXE	SmartDrive disk cache.
NE	GDI.EXE	Graphical device interface API.
NE	RECORDER.DLL	Windows recorder.
NE	VBRUN300.DLL	Visual Basic run-time DLL.

**Table 1:** Windows 3.1 programs and their exe formats.

Format	Name	Description
LX	PMSHELL.EXE	Presentation Manager Shell.
LX	CMD.EXE	Command line shell.
NE	LINK386.EXE	Linker.
NE	RC.EXE	Resource compiler.
LX	DOSCALL1.DLL	System call DLL.
LX	VIOCALLS.DLL	VIO character mode video API.

**Table 2:** Some OS/2 2.1 programs and their exe types.

Type	Target	Source
...		
Internal	1.0036	2.02f0
Internal	1.002c	2.0270
Internal	1.0022	2.01f0
...		

*Target and source specified as Object.Offset.*

**Table 3:** LX Fixup records for array references

Table Name	File location	Termination	Entry size
Segment	<i>lfanew + segtab</i>	Item count cseg.	Fixed
Entry	<i>lfanew + enttab</i>	Next file section reached.	Variable/bundled.
Resident Name	<i>lfanew + restab</i>	Null terminator.	Variable/string.
Non-Resident Name	<i>nrestab</i>	Byte count cbnres.	Variable/string.
Module Reference	<i>lfanew + modtab</i>	Item count cmod.	Fixed
Resource	<i>lfanew + rsrctab</i>	Item count cres.	????
Imported Name	<i>lfanew + imptab</i>	Null terminator.	Variable/string.
Fixup Record	<i>After corresponding segment.</i>	Item count*.	Fixed

*All symbolic references are to the NE exe header structure of listing X except lfanew which is a member of the MZ header.*

*\* Item count is 1st 2 bytes of table.*

**Table 4:** NE tables, their file locations and termination.

Table Name	File location	Termination	Entry size
Object	<i>Ifanew + ObjTbIOfs</i>	Item count - NumObjs	Fixed
Object Page	<i>Ifanew + ObjPgTbIOfs</i>	Item count*.	Fixed
Resource	<i>Ifanew + RscTbIOfs</i>	Item count - NumRscEntries	Fixed
Resident Name	<i>Ifanew + ResTbIOfs</i>	Null terminator.	Variable/string.
Non-Resident Name	<i>NResTbIOfs</i>	Byte count - NResNmTbILen	Variable/string.
Entry	<i>Ifanew + rEntryTbIOfs</i>	????	Variable/bundled.
Module Format Directives	<i>Ifanew + ModFmtTbIOfs</i>	Item count - NumModEntries.	????
Fixup Page	<i>Ifanew + FixupPgTbIOfs</i>	<i>See Object Page Table</i>	Fixed
Fixup Record	<i>Ifanew + FixupRecTbIOfst.</i>	File offset < <i>Ifanew +</i> <i>ImpModTbIOfs.</i>	Variable
Import Module Name	<i>Ifanew + ImpModTbIOfs</i>	Item count - ImpModEntries.	Variable/string.
Import Procedure Name	<i>Ifanew + ImpProcTbIOfs</i>	File offset < <i>DataPgOfs</i>	Variable/string.

*All symbolic references are to the LX exe header structure of listing X except Ifanew which is a member of the MZ header.*

**Table 5:** LX tables, their file locations and termination.

#### Segment 5

Start	Links	End	#	
0x00f2		0x00f2 0242	001	THIS_EXE.2.0000
0x00f4		0x00f4 0244	001	THIS_EXE.3.0000
0x00f6		0x00f6 0246	001	THIS_EXE.4.0000

**Table 6:** NE relocation list for listing 6.

Type	Target	Source
Internal	1.0054	2.000401ec
Internal	1.004a	2.000401e8
Internal	1.0040	2.000201f0
Internal	1.0036	2.000201ec
Internal	1.002c	2.01f4
Internal	1.0022	2.01f0

**Table 7:** LX relocation list for listing 6.

#### Segment 1

Start	Links	End	#	
0x0012	0x001b	0x0024 0036	003	DOSCALLS.0032

**Table 8:** NE relocation list for listing 7.

#### Fixups

Type	Target	Source
Import by ordinal	1.37	DOSCALLS.229 Additive = 0
Import by ordinal	1.2d	DOSCALLS.229 Additive = 0
Import by ordinal	1.23	DOSCALLS.229 Additive = 0

**Table 9:** LX relocation list for listing 7.

**SEGMENT LIST**

#	Sector	Frame	Flags	MinAlloc	Type	PL
01	0001	07f2	0d00	07f2	Code	03 (Impure) (Load on call)
02	0000	0000	0c01	fffe	Data	03 (Impure) (Load on call)
03	0000	0000	0c01	fffe	Data	03 (Impure) (Load on call)
04	0006	0203	0d01	0210	Data	03 (Impure) (Load on call)

**Table 10:** NE segments for listing 2.

**Object 1**, Size 1598, Addr 0, Flags 2005, PgTableInd 1, NumPgs 1, Rsv0

**READ, EXECUTE, 32-BIT,**

PAGES:	Number	Offset	Size	Flags
	1	0	2048	Legal Physical Page - 0000 (0x0)

**Object 2**, Size 262808, Addr 0, Flags 2003, PgTableInd 2, NumPgs 65, Rsv0

**READ, WRITE, 32-BIT,**

PAGES:	Number	Offset	Size	Flags
	2	4	512	Legal Physical Page - 0000 (0x4)
	3	0	0	Zero Filled Page - 0003 (0x0)
	4	0	0	Zero Filled Page - 0003 (0x0)
	5	0	0	Zero Filled Page - 0003 (0x0)
	6	0	0	Zero Filled Page - 0003 (0x0)
	7	0	0	Zero Filled Page - 0003 (0x0)
	8	0	0	Zero Filled Page - 0003 (0x0)
	9	0	0	Zero Filled Page - 0003 (0x0)
	10	0	0	Zero Filled Page - 0003 (0x0)
	11	0	0	Zero Filled Page - 0003 (0x0)
	12	0	0	Zero Filled Page - 0003 (0x0)
	13	0	0	Zero Filled Page - 0003 (0x0)
	14	0	0	Zero Filled Page - 0003 (0x0)
	15	0	0	Zero Filled Page - 0003 (0x0)
	16	0	0	Zero Filled Page - 0003 (0x0)
	17	0	0	Zero Filled Page - 0003 (0x0)
	18	0	0	Zero Filled Page - 0003 (0x0)
	19	0	0	Zero Filled Page - 0003 (0x0)

20	0 (0x0)	0	Zero Filled Page - 0003
21	0 (0x0)	0	Zero Filled Page - 0003
22	0 (0x0)	0	Zero Filled Page - 0003
23	0 (0x0)	0	Zero Filled Page - 0003
24	0 (0x0)	0	Zero Filled Page - 0003
25	0 (0x0)	0	Zero Filled Page - 0003
26	0 (0x0)	0	Zero Filled Page - 0003
27	0 (0x0)	0	Zero Filled Page - 0003
28	0 (0x0)	0	Zero Filled Page - 0003
29	0 (0x0)	0	Zero Filled Page - 0003
30	0 (0x0)	0	Zero Filled Page - 0003
31	0 (0x0)	0	Zero Filled Page - 0003
32	0 (0x0)	0	Zero Filled Page - 0003
33	0 (0x0)	0	Zero Filled Page - 0003
34	0 (0x0)	0	Zero Filled Page - 0003
35	0 (0x0)	0	Zero Filled Page - 0003
36	0 (0x0)	0	Zero Filled Page - 0003
37	0 (0x0)	0	Zero Filled Page - 0003
38	0 (0x0)	0	Zero Filled Page - 0003
39	0 (0x0)	0	Zero Filled Page - 0003
40	0 (0x0)	0	Zero Filled Page - 0003
41	0 (0x0)	0	Zero Filled Page - 0003
42	0 (0x0)	0	Zero Filled Page - 0003
43	0 (0x0)	0	Zero Filled Page - 0003
44	0 (0x0)	0	Zero Filled Page - 0003
45	0 (0x0)	0	Zero Filled Page - 0003
46	0 (0x0)	0	Zero Filled Page - 0003
47	0 (0x0)	0	Zero Filled Page - 0003

48	0 (0x0)	0	Zero Filled Page - 0003
49	0 (0x0)	0	Zero Filled Page - 0003
50	0 (0x0)	0	Zero Filled Page - 0003
51	0 (0x0)	0	Zero Filled Page - 0003
52	0 (0x0)	0	Zero Filled Page - 0003
53	0 (0x0)	0	Zero Filled Page - 0003
54	0 (0x0)	0	Zero Filled Page - 0003
55	0 (0x0)	0	Zero Filled Page - 0003
56	0 (0x0)	0	Zero Filled Page - 0003
57	0 (0x0)	0	Zero Filled Page - 0003
58	0 (0x0)	0	Zero Filled Page - 0003
59	0 (0x0)	0	Zero Filled Page - 0003
60	0 (0x0)	0	Zero Filled Page - 0003
61	0 (0x0)	0	Zero Filled Page - 0003
62	0 (0x0)	0	Zero Filled Page - 0003
63	0 (0x0)	0	Zero Filled Page - 0003
64	0 (0x0)	0	Zero Filled Page - 0003
65	0 (0x0)	0	Zero Filled Page - 0003
66	0 (0x0)	0	Zero Filled Page - 0003

**Object 3**, Size 49152, Addr 0, Flags 2003, PgTableInd 67, NumPgs 1, Rsv0

<b>READ, WRITE, 32-BIT,</b>				
<b>PAGES:</b>	<b>Number</b>	<b>Offset</b>	<b>Size</b>	<b>Flags</b>
	67	0 (0x0)	0	Zero Filled Page - 0003

**Table 11:** LX Object and Object Page Tables for Listing 2.

## LISTING 1 IS A LONG LISTING CONTAINED IN THE FILE *RODLEY.LS1*.

```
#include <stdio.h>
int array1[32767], array2[32767];
int main() { array1[0] = 1; array2[0] = 2; return( 0 ); }
```

**Listing 2:** Simple program with 2 large arrays and 2 array reference

```
#include <stdio.h>
int array1[32], array2[32], array3[32];
int main() { array1[0] = 7; array2[0] = 8; array3[0] = 9; return( 0 ); }
```

**Listing 4:** Simple program with 3 small arrays and 3 array reference

```
BCTEST1.C#11      array1[0] = 7;
11      0x00010020 C705 F0010200 070000 MOV DWORD PTR [array1], 00000007
BCTEST1.C#12      array2[0] = 8;
12      0x0001002A C705 70020200 080000 MOV DWORD PTR [array2], 00000008
BCTEST1.C#13      array3[0] = 9;
13      0x00010034 C705 F0020200 090000 MOV DWORD PTR [array3], 00000009
```

Source offset = 0x000202F0

**Listing 5:** Disassembly of LX array references

```
#include <stdio.h>
int array1[32767], array2[32767], array3[32767];
int main() { array1[0] = 1; array1[1] = 2;
            array2[0] = 1; array2[1] = 2;
            array3[0] = 1; array3[1] = 2; return( 0 ); }
```

**Listing 6:** 3 large arrays and 2 references to each array

```
#include <os2.h>
#define INCL_DOSPROCESS
int main() { DosSleep( 2 ); DosSleep( 2 ); DosSleep( 2 );
            return( 0 ); }
```

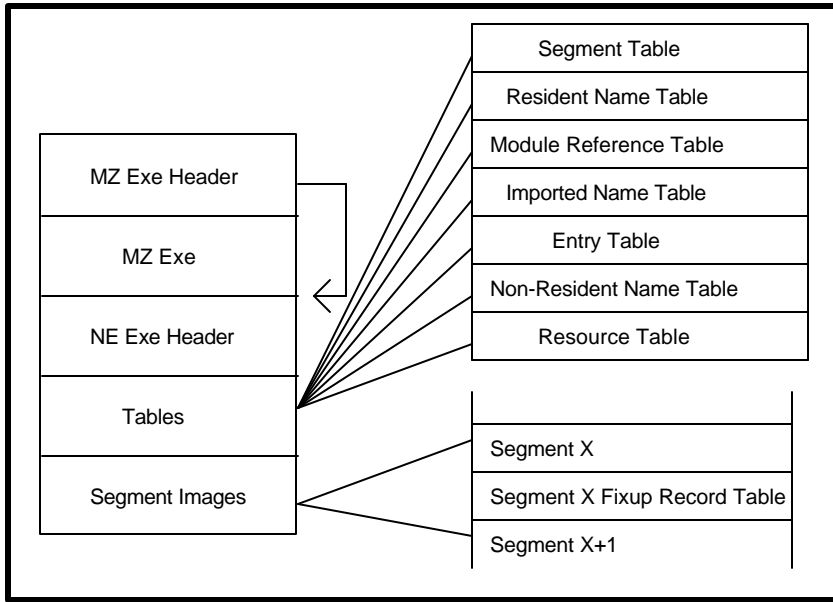
**Listing 7:** Program with 3 external fixups.

```
000F:0001 8BEC      MOV     BP,SP
000F:0003 B80000      MOV     AX,0000
000F:0006 9A72020F00 CALL    000F:0272
000F:000B 57          PUSH   DI
000F:000C 56          PUSH   SI
7:      DosSleep( 2 ); DosSleep( 2 ); DosSleep( 2 );
000F:000D 6A00      PUSH   00
000F:000F 6A02      PUSH   02
```

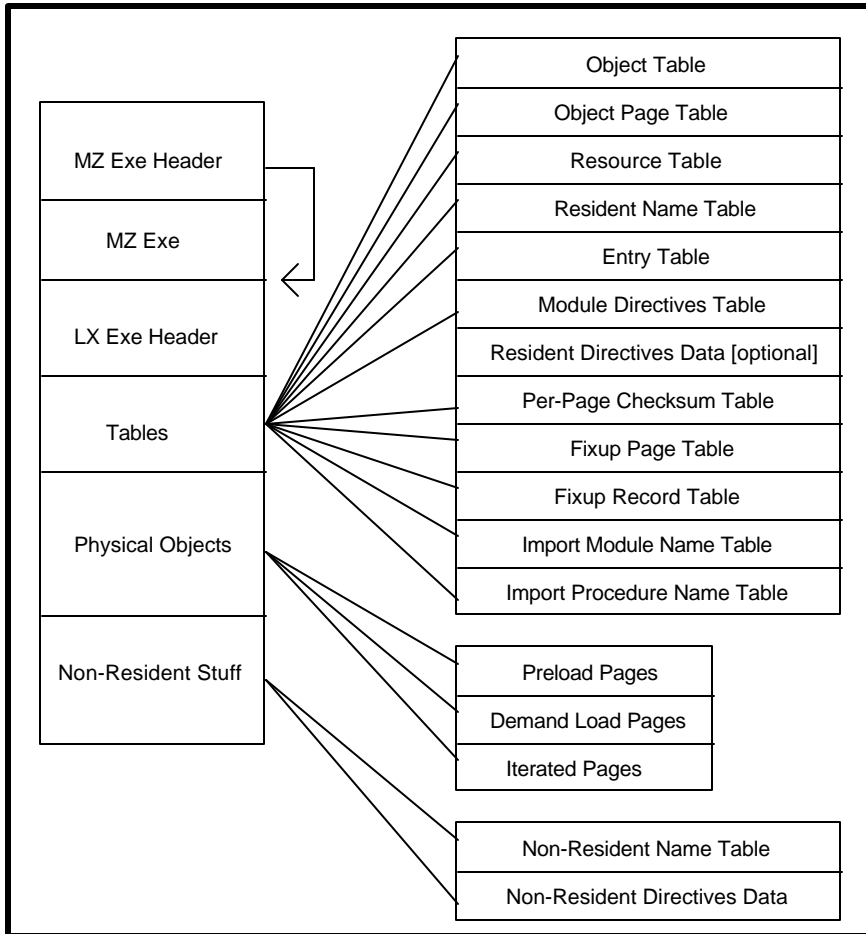
```
000F:0011 9A1B000000 CALL 0000:001B
000F:0016 6A00 PUSH 00
000F:0018 6A02 PUSH 02
000F:001A 9A24000000 CALL 0000:0024
000F:001F 6A00 PUSH 00
000F:0021 6A02 PUSH 02
000F:0023 9AFF000000 CALL 0000:FFFF
8: return( 0 ); }
000F:0028 B80000 MOV AX,0000
000F:002B E90000 JMP 002E
000F:002E 5E POP SI
000F:002F 5F POP DI
000F:0030 C9 LEAVE
```

*[If you look at this through CODEVIEW, the target chain links will be shown as real addresses. You have to look at the file on disk to see the chain of targets.]*

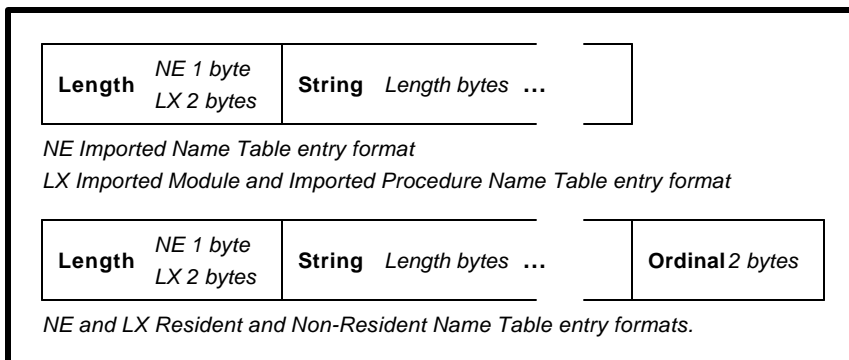
**Listing 8:** *NE Disassembly of Listing 7.*



**Figure 1:** NE Physical Map.



**Figure 2:** LX Physical Map.



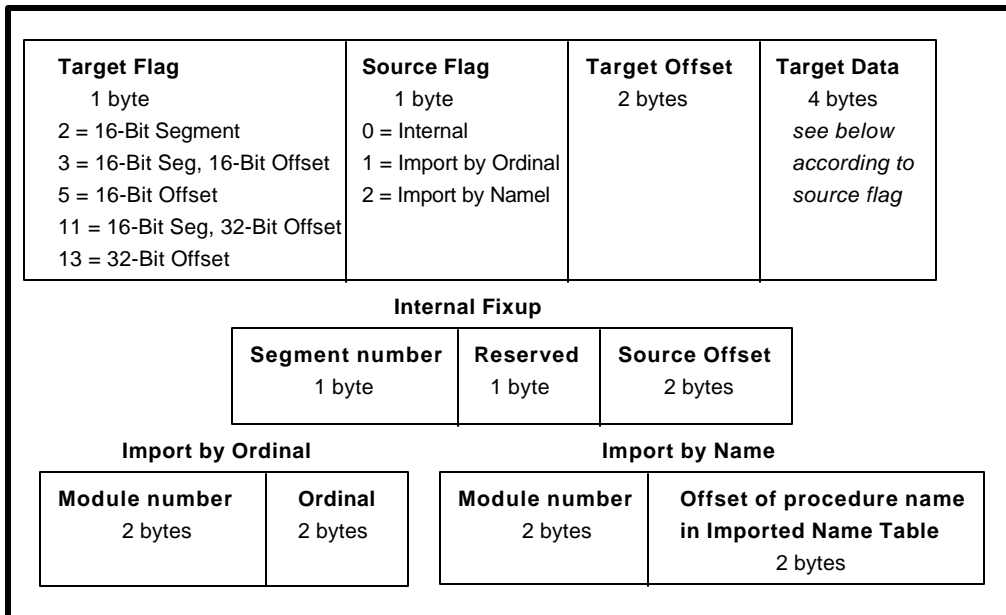
**Figure 3:** LX and NE string table entry formats.

<b>Count</b> 1 byte  If Type = unused increment ordinal by this amount.	<b>Type</b> 1 byte  0 - unused 0xFF -Movable segment recs nn - Fixed segment recs for segment nn.	<b>Bundle ...</b> Count occurrences of Movable or Fixed Segment Record  Does not occur if Type = unused.
<b>Movable Segment Record</b>		<b>Fixed Segment Record</b>
<b>Flags</b> 1 byte  1 - Exported Entry 2 - Uses Shared Data	<b>Reserved</b> 2 bytes 0xCD3F	<b>Segment #</b> 1 byte
		<b>Offset</b> 2 bytes
		<b>Flags</b> 1 byte  1 - Exported Entry 2 - Uses Shared Data
		<b>Offset</b> 2 bytes

Figure 4: NE Entry Table entry format.

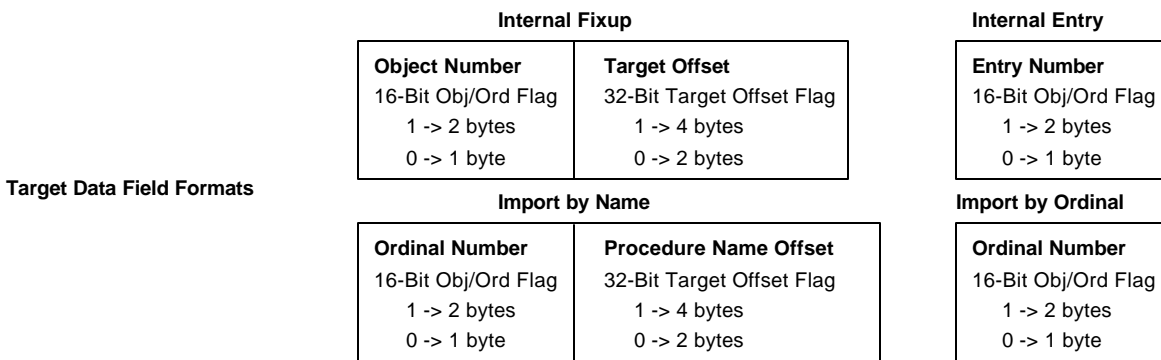
<b>Count</b> 1 byte  If Type = unused increment ordinal by this amount.	<b>Type</b> 1 byte  bits 0-3 Type 0 - unused 1 - 16-Bit 2 - 286 Call Gate 3 - 32-Bit 4 - Forwarder	<b>Bundle ...</b>  Does not occur if Type = unused.
<b>16-Bit</b>		<b>32-Bit</b>
<b>Flags</b> 1 byte	<b>Offset</b> 2 bytes	<b>Object</b> 2 bytes
		<b>Flags</b> 1 byte
		<b>Offset</b> 4 bytes
<b>286 Call Gate</b>		
<b>Object</b> 2 bytes	<b>Flags</b> 1 byte	<b>Offset</b> 2 bytes
		<b>Call Gate</b> 2 bytes
<b>Forwarder</b>		
<b>Reserved</b> 2 bytes	<b>Flags</b> 1 byte	<b>Module Ordinal #</b> 2 bytes
		<b>Proc Name Offset/Ordinal #</b> 4 bytes

Figure 5: LX Entry Table entry format.

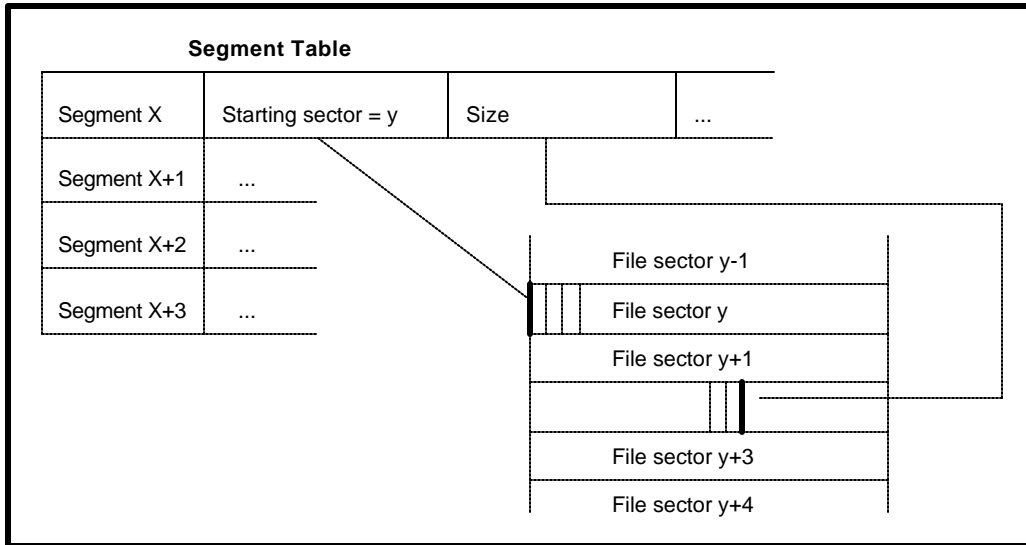


**Figure 6:** NE Fixup Record Format

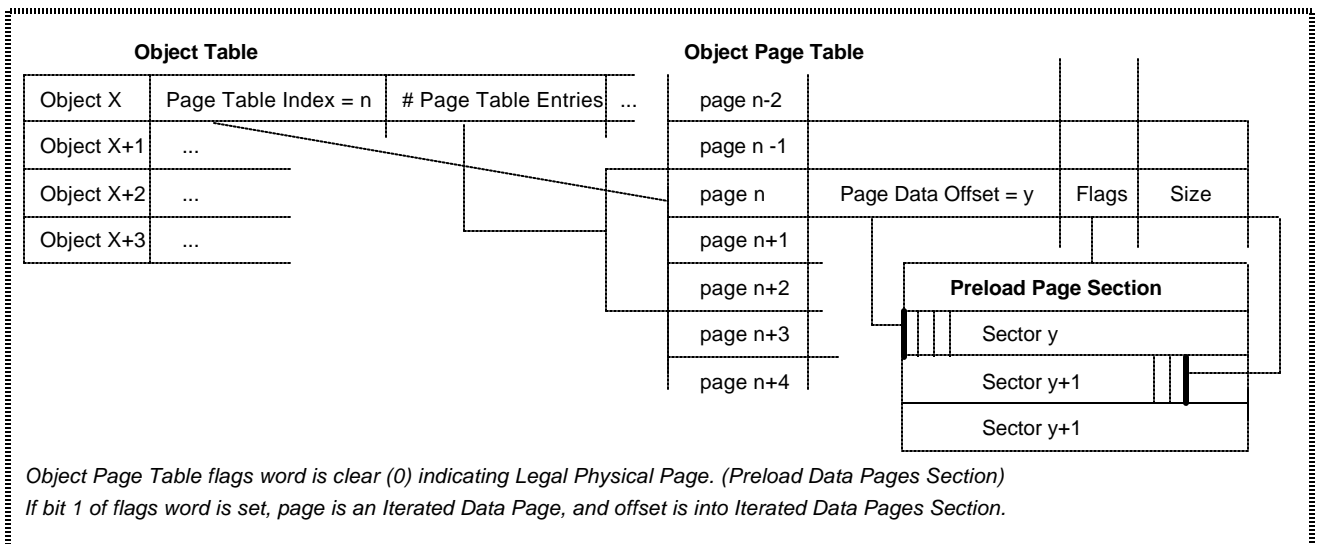
<b>Source Flags</b> 1 byte bits 0-3 = Source Type 0 = 8-Bit 2 = 16-Bit Selector 3 = 16 5 = 16-Bit Offset 6 = 16 7 = 32-Bit Offset bit 4 = 32-Bit Self-rel bit 5 = Fixup to Alias bit 6 = Source List	<b>Target Flags</b> 1 byte bits 0-1 = Target Type 0 = Internal 1 = Import by ordinal 2 = Import by name 3 = Internal Entry bit 2 = Additive bit 4 = 32-Bit Target bit 5 = 32-Bit Additive bit 6 = 16-Bit Obj/Ord bit 7 = 8-Bit Ordinal	<b>Source List Flag</b> 1 -> List Count. 0 -> Offset.  <b>Source Offset</b> 2 byte  <b>Source List Count</b> 1 byte	<b>Target Data</b>   <i>see below by Target Type</i>	<b>Additive</b> 32-Bit Additive Flag 1 -> 4 bytes 0 -> 2 bytes  Additive Flag 1 -> List exists. 0 -> No list.  <i>Not available for Target Type = Internal</i>	<b>Source List</b> Source List Count entries at 2 bytes each.  Source List Flag 1 -> List exists. 0 -> No list.
---	---	---	---	---	--



**Figure 7:** LX Fixup Record Format



**Figure 8:** NE Segment Table and Segments  
a



**Figure 9:** LX Object Table, Object Page Table and Physical Pages