IBM OS/2 16/32-bit Linear eXecutable Module Format (LX)

Revision 11

June 14, 2001 11:00 am

IBM
Operating System/2
OS/2

Contents

---

1 Introduction

This document describes the IBM OS/2 16/32-bit Linear eXecutable Module Format (LX). This is

the load module format understood by the OS/2 32-bit system loader (for OS/2 version 2.0 and greater). LX load modules are created by Linear Executable linker utilities (such as IBM LINK386). A Linear Executable linker must be used in order to create 32-bit (flat-model) OS/2 programs; however, the LX format also allows for any combination of 16-bit and 32-bit code or data sections to exist within the same module.

---

2 Linear Executable Module Format

The following sections describe the Linear Executable Module Format in detail.

2.1 Linear Executable Sections

The LX module header may optionally be preceded by a DOS 2 compatible module header. The DOS 2 compatible module header is identified by the first two bytes of the header containing the signature characters "MZ". If the DOS 2 compatible module header is absent then the module will start with the LX module header.

If a module begins with a DOS 2 compatible module header, then the following technique should be used to locate the LX module header, if present. The word at offset 18h in the DOS 2 compatible module header is the offset to the DOS relocation table. If this offset is 40h, then the doubleword at offset 3Ch in the DOS 2 compatible module header contains the offset, from the beginning of the file, to the new module header. This may be an LX module header and can be identified by the first two bytes of the header containing the signature characters "LX". If a valid module header is not found, then the file is a DOS 2 compatible module.

The remainder of the DOS 2 compatible module header will describe a DOS stub program. The stub may be any valid program but will typically be a program which displays an error message. It may also be a DOS version of the LX program.

Figure 3-1: **DOS 2.0 Section (**Discarded after processed by the loader)

00h	DOS 2 Compatible EXE header
1Ch	unused
24h	OEM Identifier
26h	OEM Info
3Ch	Offset to Linear EXE Header
40h	DOS 2.0 Stub Program and Relocation Table

Figure 3-2: Linear Executable

Module Header (In-memory copy maintained)

xxh	Executable Information
	Module Information
	Loader Section Information
	Table Offset Information

Figure 3-3: Loader Section (In-memory copy maintained)

Object Table
Object Page Table
Resource Table
Resident Name Table
Entry Table
Module Format Directives Table (Optional)
Resident Directives Data (Optional) (Verify Record)
Per-Page Checksum

Figure 3-4: Fixup Section (In-memory copy maintained)

Fixup Page Table
Fixup Record Table
Import Module Name Table
Import Procedure Name Table

The Fixup Section must immediately follow the Loader Section in the executable file. Although they are defined as two separate sections, the OS/2 loader currently treats these two sections as one section.

Figure 3-5: Data Section (No In-memory copy maintained)

Preload Pages
Demand Load Pages
Iterated Pages

Non-Resident Name Table
Non-Resident Directives Data (Optional) (To be Defined)

Figure 3-6: Debug Section (Not used by the Loader)

Debugger Information

Note: The standard section ordering of an LX module is the LX module header, the resident sections, the non-resident sections and finally the debug section (if present). It is also permissable to use an alternate section ordering of the LX module header, the non-resident sections, the resident sections and finally the debug section (if present).

2.2 LX Header Fields

Figure 3-7: 32-bit Linear EXE Header

00h	"L" "X"	B-ORD	W-ORD	FORMAT LEVEL
08h	CPU TYPE	OS TYPE		MODULE VERSION
10h	MODULE FLAGS			MODULE # OF PAGES
18h	EIP OBJECT #			EIP
20h	ESP OBJECT #			ESP
28h	PAGE SIZE			PAGE OFFSET SHIFT
30h	FIXUP SECTION SIZE			FIXUP SECTION CHECKSUM
38h	LOADER SECTION SIZE			LOADER SECTION CHECKSUM
40h	OBJECT TABLE OFF			# OBJECTS IN MODULE
48h	OBJECT PAGE TABLE OFF			OBJECT ITER PAGES OFF
50h	RESOURCE TABLE OFFSET			#RESOURCE TABLE ENTRIES
58h	RESIDENT NAME TBL OFF			ENTRY TABLE OFFSET
60h	MODULE DIRECTIVES OFF			# MODULE DIRECTIVES
68h	FIXUP PAGE TABLE OFF			FIXUP RECORD TABLE OFF
70h	IMPORT MODULE TBL OFF			# IMPORT MOD ENTRIES
78h	IMPORT PROC TBL OFF			PER-PAGE CHECKSUM OFF
80h	DATA PAGES OFFSET			#PRELOAD PAGES

88h	NON-RES NAME TBL OFF	NON-RES NAME TBL LEN
90h	NON-RES NAME TBL CKSM	AUTO DS OBJECT #
98h	DEBUG INFO OFF	DEBUG INFO LEN
A0h	#INSTANCE PRELOAD	#INSTANCE DEMAND
A8h	HEAPSIZE	STACKSIZE

Note: The OBJECT ITER PAGES OFF must either be 0 or set to the same value as DATA PAGES OFFSET in OS/2 2.0. I.e. iterated pages are required to be in the same section of the file as regular pages.

Note: Table offsets in the Linear EXE Header may be set to zero to indicate that the table does not exist in the EXE file and it's size is zero.

"L" "X" = DW Signature word.

The signature word is used by the loader to identify the EXE file as a valid 32-bit Linear Executable Module Format. "L" is low order byte. "X" is high order byte.

B-ORD = DB Byte Ordering.

This byte specifies the byte ordering for the linear EXE format. The values are:

00H - Little Endian Byte Ordering.
01H - Big Endian Byte Ordering.

W-ORD = DB Word Ordering.

This byte specifies the Word ordering for the linear EXE format. The values are:

00H - Little Endian Word Ordering.
01H - Big Endian Word Ordering.

Format Level = DD Linear EXE Format Level.

The Linear EXE Format Level is set to 0 for the initial version of the 32-bit linear EXE format. Each incompatible change to the linear EXE format must increment this value. This allows the system to recognized future EXE file versions so that an appropriate error message may be displayed if an attempt is made to load them.

CPU Type = DW Module CPU Type.

This field specifies the type of CPU required by this module to run. The values are:

01H - 80286 or upwardly compatible CPU is required to execute this module.
02H - 80386 or upwardly compatible CPU is required to execute this module.
03H - 80486 or upwardly compatible CPU is required to execute this module.

OS Type = DW Module OS Type.

This field specifies the type of Operating system required to run this module. The currently defined values are:

00H - Unknown (any "new-format" OS)
01H - OS/2 (default)
02H - Windows[1]
03H - DOS 4.x
04H - Windows 386
05H - IBM Microkernel Personality Neutral

MODULE VERSION = DD Version of the linear EXE module.

This is useful for differentiating between revisions of dynamic linked modules. This value is specified at link time by the user.

MODULE FLAGS = DD Flag bits for the module.

The module flag bits have the following definitions.

00000001h = Reserved for system use.
00000002h = Reserved for system use.
00000004h = Per-Process Library Initialization.

The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Initialization is assumed. Setting this bit for an EXE file is invalid.

00000008h = Reserved for system use.
00000010h = Internal fixups for the module have been applied.

The setting of this bit in a Linear Executable Module indicates that each object of the module has a preferred load address specified in the Object Table Reloc Base Addr. If the module's objects can not be loaded at these preferred addresses, then the relocation records that have been retained in the file data will be applied.

00000020h = External fixups for the module have been applied.
00000040h = Reserved for system use.
00000080h = Reserved for system use.
00000100h = Incompatible with PM windowing.
00000200h = Compatible with PM windowing.
00000300h = Uses PM windowing API.
00000400h = Reserved for system use.
00000800h = Reserved for system use.

00001000h = Reserved for system use.
00002000h = Module is not loadable.

When the 'Module is not loadable' flag is set, it indicates that either errors were detected at link time or that the module is being incrementally linked and therefore can't be loaded.

00004000h = Reserved for system use.
00038000h = Module type mask.
00000000h = Program module (EXE).

A module can not contain dynamic links to other modules that have the 'program module' type.

00008000h = Library module (DLL).
00010000h = Reserved for system use.
00018000h = Reserved for system use.
00020000h = Physical Device Driver module.
00028000h = Virtual Device Driver module.
00030000h = DLD module.
00038000h = Reserved for system use.
00080000h = MP-unsafe.

The program module is multiple-processor unsafe. It does not provide the necessary serialization to run on more than one CPU at a time.

40000000h = Per-process Library Termination.

The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Termination is assumed. Setting this bit for an EXE file is invalid.

MODULE # PAGES = DD Physical number of pages in module.

This field specifies the number of pages physically contained in this module. In other words, pages containing either enumerated or iterated data, not invalid or zero-fill pages. These pages are contained in the 'preload pages', 'demand load pages' and 'iterated data pages' sections of the linear EXE module. This is used to determine the size of the other physical page based tables in the linear EXE module.

EIP OBJECT # = DD The Object number to which the Entry Address is relative.

This specifies the object to which the Entry Address is relative. This must be a nonzero value for a program module to be correctly loaded. A zero value for a library module indicates that no library entry routine exists. If this value is zero, then both the Per-process Library Initialization bit and the Per-process Library Termination bit must

be clear in the module flags, or else the loader will fail to load the module. Further, if the Per-process Library Termination bit is set, then the object to which this field refers must be a 32-bit object (i.e., the Big/Default bit must be set in the object flags; see below).

EIP = DD Entry Address of module.

The Entry Address is the starting address for program modules and the library initialization and Library termination address for library modules.

ESP OBJECT # = DD The Object number to which the ESP is relative.

This specifies the object to which the starting ESP is relative. This must be a nonzero value for a program module to be correctly loaded. This field is ignored for a library module.

ESP = DD Starting stack address of module.

The ESP defines the starting stack pointer address for program modules. A zero value in this field indicates that the stack pointer is to be initialized to the highest address/offset in the object. This field is ignored for a library module.

PAGE SIZE = DD The size of one page for this system.

This field specifies the page size used by the linear EXE format and the system. For the initial version of this linear EXE format the page size is 4Kbytes. (The 4K page size is specified by a value of 4096 in this field.)

PAGE OFFSET SHIFT = DD The shift left bits for page offsets.

This field gives the number of bit positions to shift left when interpreting the Object Page Table entries' page offset field. This determines the alignment of the page information in the file. For example, a value of 4 in this field would align all pages in the Data Pages and Iterated Pages sections on 16 byte (paragraph) boundaries. A Page Offset Shift of 9 would align all pages on a 512 byte (disk sector) basis. All other offsets are byte aligned.
A page might not start at the next available alignment boundary. Extra padding is acceptable between pages as long as each page starts on an alignment boundary. For example, several alignment boundaries may be skipped in order to start a frequently accessed page on a sector boundary.

FIXUP SECTION SIZE = DD Total size of the fixup information in bytes.

This includes the following 4 tables:

Fixup Page Table
Fixup Record Table
Import Module name Table

Import Procedure Name Table

FIXUP SECTION CHECKSUM = DD Checksum for fixup information.

This is a cryptographic checksum covering all of the fixup information. The checksum for the fixup information is kept separate because the fixup data is not always loaded into main memory with the 'loader section'. If the checksum feature is not implemented, then the linker will set these fields to zero.

LOADER SECTION SIZE = DD Size of memory resident tables.

This is the total size in bytes of the tables required to be memory resident for the module, while the module is in use. This total size includes all tables from the Object Table down to and including the Per-Page Checksum Table.

LOADER SECTION CHECKSUM = DD Checksum for loader section.

This is a cryptographic checksum covering all of the loader section information. If the checksum feature is not implemented, then the linker will set these fields to zero.

OBJECT TABLE OFF = DD Object Table offset.

This offset is relative to the beginning of the linear EXE header. This offset also points to the start of the Loader Section.

OBJECTS IN MODULE = DD Object Table Count.

This defines the number of entries in Object Table.

OBJECT PAGE TABLE OFFSET = DD Object Page Table offset

This offset is relative to the beginning of the linear EXE header.

OBJECT ITER PAGES OFF = DD Object Iterated Pages offset.

This offset is relative to the beginning of the EXE file.

RESOURCE TABLE OFF = DD Resource Table offset.

This offset is relative to the beginning of the linear EXE header.

RESOURCE TABLE ENTRIES = DD Number of entries in Resource Table.
RESIDENT NAME TBL OFF = DD Resident Name Table offset.

This offset is relative to the beginning of the linear EXE header.

ENTRY TBL OFF = DD Entry Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES OFF = DD Module Format Directives Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES = DD Number of Module Format Directives in the Table.

This field specifies the number of entries in the Module Format Directives Table.

FIXUP PAGE TABLE OFF = DD Fixup Page Table offset.

This offset is relative to the beginning of the linear EXE header. This offset also points to the start of the Fixup Section.

FIXUP RECORD TABLE OFF = DD Fixup Record Table Offset

This offset is relative to the beginning of the linear EXE header.

IMPORT MODULE TBL OFF = DD Import Module Name Table offset.

This offset is relative to the beginning of the linear EXE header.

IMPORT MOD ENTRIES = DD The number of entries in the Import Module Name Table.
IMPORT PROC TBL OFF = DD Import Procedure Name Table offset.

This offset is relative to the beginning of the linear EXE header.

PER-PAGE CHECKSUM OFF = DD Per-Page Checksum Table offset.

This offset is relative to the beginning of the linear EXE header.

DATA PAGES OFFSET = DD Data Pages Offset.

This offset is relative to the beginning of the EXE file. This offset also points to the start of the Data Section.

PRELOAD PAGES = DD Number of Preload pages for this module.

Note: OS/2 2.0 does not respect the preload of pages as specified in the executable file for performance reasons.

NON-RES NAME TBL OFF = DD Non-Resident Name Table offset.

This offset is relative to the beginning of the EXE file.

NON-RES NAME TBL LEN = DD Number of bytes in the Non-resident name table.
NON-RES NAME TBL CKSM = DD Non-Resident Name Table Checksum.

This is a cryptographic checksum of the Non-Resident Name Table.

AUTO DS OBJECT # = DD The Auto Data Segment Object number.

This is the object number for the Auto Data Segment used by 16-bit modules. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

DEBUG INFO OFF = DD Debug Information offset.

This offset is relative to the beginning of the file. This offset also points to the start of the Debug Section.

Note: Earlier versions of this document stated that this offset was from the linear EXE header - this is incorrect.

DEBUG INFO LEN = DD Debug Information length.

The length of the debug information in bytes.

INSTANCE PRELOAD = DD Instance pages in preload section.

The number of instance data pages found in the preload section.

INSTANCE DEMAND = DD Instance pages in demand section.

The number of instance data pages found in the demand section.

HEAPSIZE = DD Heap size added to the Auto DS Object.

The heap size is the number of bytes added to the Auto Data Segment by the loader. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

STACKSIZE = DD Stack size.

The stack size is the number of bytes specified by:

1. size of a segment with combine type stack
2. STACKSIZE in the .DEF file
3. /STACK link option

The stacksize may be zero.

Note: Stack sizes with byte 2 equal to 02 or 04 (e.g. 00020000h, 11041111h, 0f02ffffh) should be avoided for programs that will run on OS/2 2.0.

2.3 Program (EXE) startup registers and Library entry registers

Program startup registers are defined as follows.

EIP = Starting program entry address.
ESP = Top of stack address.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Data selector of base of Thread Information Block (TIB).
GS = 0.
EAX = EBX = 0.
ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.
[ESP+0] = Return address to routine which calls DosExit(1,EAX).
[ESP+4] = Module handle for program module.
[ESP+8] = Reserved.
[ESP+12] = Environment data object address.
[ESP+16] = Command line linear address in environment data object.

Library initialization registers are defined as follows.

EIP = Library entry address.
ESP = User program stack.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Data selector of base of Thread Information Block (TIB).
GS = 0.
EAX = EBX = 0.
ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.
[ESP+0] = Return address to system, (EAX) = return code.
[ESP+4] = Module handle for library module.
[ESP+8] = 0 (Initialization)

Note: A 32-bit library may specify that its entry address is in a 16-bit code object. In this case, the entry registers are the same as for entry to a library using the Segmented EXE format. These are documented elsewhere. This means that a 16-bit library may be relinked to take advantage of the benefits of the Linear EXE format (notably, efficient paging).

Library termination registers are defined as follows.

EIP = Library entry address.
ESP = User program stack.
CS = Code selector for base of linear address space.
DS = ES = SS = Data selector for base of linear address space.
FS = Data selector of base of Thread Information Block (TIB).
GS = 0.
EAX = EBX = 0.

ECX = EDX = 0.
ESI = EDI = 0.
EBP = 0.
[ESP+0] = Return address to system.
[ESP+4] = Module handle for library module.
[ESP+8] = 1 (Termination)

Note: Library termination is not allowed for libraries with 16-bit entries.

2.4 Object Table

The number of entries in the Object Table is given by the # Objects in Module field in the linear EXE header. Entries in the Object Table are numbered starting from one.

Each Object Table entry has the following format:

Figure 3-8: Object Table

00h	VIRTUAL SIZE	RELOC BASE ADDR
08h	OBJECT FLAGS	PAGE TABLE INDEX
10h	# PAGE TABLE ENTRIES	RESERVED

VIRTUAL SIZE = DD Virtual memory size.

This is the size of the object that will be allocated when the object is loaded. The object data length must be less than or equal to the total size of the pages in the EXE file for the object. This memory size must also be large enough to contain all of the iterated data and uninitialized data in the EXE file.

RELOC BASE ADDR = DD Relocation Base Address.

The relocation base address the object is currently relocated to. If the internal relocation fixups for the module have been removed, this is the address the object will be allocated at by the loader.

OBJECT FLAGS = DW Flag bits for the object.

The object flag bits have the following definitions.

0001h = Readable Object.
0002h = Writable Object.
0004h = Executable Object.

The readable, writable and executable flags provide support for all

possible protections. In systems where all of these protections are not supported, the loader will be responsible for making the appropriate protection match for the system.

0008h = Resource Object.
0010h = Discardable Object.
0020h = Object is Shared.
0040h = Object has Preload Pages.
0080h = Object has Invalid Pages.
0100h = Object is Resident (valid for VDDs, PDDs only).
0200h = Reserved.
0300h = Object is Resident & Contiguous (VDDs, PDDs only).
0400h = Object is Resident & 'long-lockable' (VDDs, PDDs only).
0800h = Object is marked as an IBM Microkernel extension.
1000h = 16:16 Alias Required (80x86 Specific).
2000h = Big/Default Bit Setting (80x86 Specific).

The 'big/default' bit, for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment descriptor. (The Default bit, or D-bit, determines whether the default word size is 32-bits or 16-bits. It also affects the interpretation of the instruction stream.)

4000h = Object is conforming for code (80x86 Specific).
8000h = Object I/O privilege level (80x86 Specific).

Only used for 16:16 Alias Objects.

PAGE TABLE INDEX = DD Object Page Table Index.

This specifies the number of the first object page table entry for this object. The object page table specifies where in the EXE file a page can be found for a given object and specifies per-page attributes.
The object table entries are ordered by logical page in the object table. In other words the object table entries are sorted based on the object page table index value.

PAGE TABLE ENTRIES = DD # of object page table entries for this object.

Any logical pages at the end of an object that do not have an entry in the object page table associated with them are handled as zero filled or invalid pages by the loader. When the last logical pages of an object are not specified with an object page table entry, they are treated as either zero filled pages or invalid pages based on the last entry in the object page table for that object. If the last entry was neither a zero filled or invalid page, then the additional pages are treated as zero filled pages.

RESERVED = DD Reserved for future use. Must be set to zero.

2.5 Object Page Table

The Object page table provides information about a logical page in an object. A logical page may be an enumerated page, a pseudo page or an iterated page. The structure of the object page table in conjunction with the structure of the object table allows for efficient access of a page when a page fault occurs, while still allowing the physical page data to be located in the preload page, demand load page or iterated data page sections in the linear EXE module. The logical page entries in the Object Page Table are numbered starting from one. The Object Page Table is parallel to the Fixup Page Table as they are both indexed by the logical page number.

Each Object Page Table entry has the following format:

Figure 3-9: Object Page Table Entry

	63 32	31 16	15 0
00h	PAGE DATA OFFSET	DATA SIZE	FLAGS

PAGE DATA OFFSET = DD Offset to the page data in the EXE file.

This field, when bit shifted left by the PAGE OFFSET SHIFT from the module header, specifies the offset from the beginning of the Preload Page section of the physical page data in the EXE file that corresponds to this logical page entry. The page data may reside in the Preload Pages, Demand Load Pages or the Iterated Data Pages sections. A page might not start at the next available alignment boundary. Extra padding is acceptable between pages as long as each page starts on an alignment boundary. For example, several alignment boundaries may be skipped in order to start a frequently accessed page on a sector boundary.
If the FLAGS field specifies that this is a Zero-Filled page then the PAGE DATA OFFSET field will contain a 0.
If the logical page is specified as an iterated data page, as indicated by the FLAGS field, then this field specifies the offset into the Iterated Data Pages section.
The logical page number (Object Page Table index), is used to index the Fixup Page Table to find any fixups associated with the logical page.

DATA SIZE = DW Number of bytes of data for this page.

This field specifies the actual number of bytes that represent the page in the file. If the PAGE SIZE field from the module header is greater than the value of this field and the FLAGS field indicates a Legal Physical Page, the remaining bytes are to be filled with zeros. If the FLAGS field indicates an Iterated Data Page, the iterated data records will completely fill out the remainder.

FLAGS = DW Attributes specifying characteristics of this logical page.

The bit definitions for this word field follow,

00h = Legal Physical Page in the module (Offset from Preload Page Section).
01h = Iterated Data Page (Offset from Iterated Data Pages Section).
02h = Invalid Page (zero).
03h = Zero Filled Page (zero).
04h = Unused.
05h = Compressed Page (Offset from Preload Pages Section).

2.6 Resource Table

The resource table is an array of resource table entries. Each resource table entry contains a type ID and name ID. These entries are used to locate resource objects contained in the Object table. The number of entries in the resource table is defined by the Resource Table Count located in the linear EXE header. More than one resource may be contained within a single object. Resource table entries are in a sorted order, (ascending, by Resource Name ID within the Resource Type ID). This allows the DosGetResource API function to use a binary search when looking up a resource in a 32-bit module instead of the linear search being used in the current 16-bit module.

Each resource entry has the following format:

Figure 3-10: Resource Table

00h	TYPE ID	NAME ID
04h	RESOURCE SIZE	
08h	OBJECT	OFFSET

TYPE ID = DW Resource type ID.

The type of resources are:

01h = RT_POINTER = mouse pointer shape
02h = RT_BITMAP = bitmap
03h = RT_MENU = menu template
04h = RT_DIALOG = dialog template
05h = RT_STRING = string tables
06h = RT_FONTDIR = font directory
07h = RT_FONT = font
08h = RT_ACCELTABLE = accelerator tables
09h = RT_RCDATA = binary data
0Ah = RT_MESSAGE = error msg tables
0Bh = RT_DLGINCLUDE = dialog include file name

0Ch = RT_VKEYTBL = key to vkey tables
0Dh = RT_KEYTBL = key to UGL tables
0Eh = RT_CHARTBL = glyph to character tables
0Fh = RT_DISPLAYINFO = screen display information
10h = RT_FKASHORT = function key area short form
11h = RT_FKALONG = function key area long form
12h = RT_HELPTABLE = Help table for Cary Help manager
13h = RT_HELPSUBTABLE = Help subtable for Cary Help manager
14h = RT_FDDIR = DBCS uniq/font driver directory
15h = RT_FD = DBCS uniq/font driver

NAME ID = DW An ID used as a name for the resource when referred to.
RESOURCE SIZE = DD The number of bytes the resource consists of.
OBJECT = DW The number of the object which contains the resource.
OFFSET = DD The offset within the specified object where the resource begins.

2.7 Resident or Non-resident Name Table Entry

The resident and non-resident name tables define the ASCII names and ordinal numbers for exported entries in the module. In addition the first entry in the resident name table contains the module name. These tables are used to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the entry table.

The resident name table is kept resident in system memory while the module is loaded. It is intended to contain the exported entry point names that are frequently dynamically linked to by name. Non-resident names are not kept in memory and are read from the EXE file when a dynamic link reference is made. Exported entry point names that are infrequently dynamically linked to by name or are commonly referenced by ordinal number should be placed in the non-resident name table. The trade off made for references by name is performance vs. memory usage.

Import references by name require these tables to be searched to obtain the entry point ordinal number. Import references by ordinal number provide the fastest lookup since the search of these tables is not required.

Installable File Systems, Physical Device Drivers, and Virtual Device Drivers are closed after the file is loaded. Any reference to the non-resident name table after this time will fail.

The strings are CASE SENSITIVE and are NOT NULL TERMINATED.

Each name table entry has the following format:

Figure 3-11: Resident or Non-resident

Name Table Entry

00h	LEN	ASCII STRING . . .	ORDINAL

LEN = DB String Length.

This defines the length of the string in bytes. A zero length indicates there are no more entries in table. The length of each ascii name string is limited to 255 characters.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

ORDINAL # = DW Ordinal number.

The ordinal number in an ordered index into the entry table for this entry point.

2.8 Entry Table

The entry table contains object and offset information that is used to resolve fixup references to the entry points within this module. Not all entry points in the entry table will be exported, some entry points will only be used within the module. An ordinal number is used to index into the entry table. The entry table entries are numbered starting from one.

The list of entries are compressed into 'bundles', where possible. The entries within each bundle are all the same size. A bundle starts with a count field which indicates the number of entries in the bundle. The count is followed by a type field which identifies the bundle format. This provides both a means for saving space as well as a mechanism for extending the bundle types.

The type field allows the definition of 256 bundle types. The following bundle types will initially be defined:

Unused Entry.
16-bit Entry.
286 Call Gate Entry.
32-bit Entry.
Forwarder Entry.

The bundled entry table has the following format:

Figure 3-12: Entry Table

00h	CNT	TYPE	BUNDLE INFO . . .

CNT = DB Number of entries.

This is the number of entries in this bundle.
A zero value for the number of entries identifies the end of the entry table. There is no further bundle information when the number of entries is zero. In other words the entry table is terminated by a single zero byte.

TYPE = DB Bundle type.

This defines the bundle type which determines the contents of the BUNDLE INFO. The follow types are defined:

00h = Unused Entry.
01h = 16-bit Entry.
02h = 286 Call Gate Entry.
03h = 32-bit Entry.
04h = Forwarder Entry.
80h = Parameter Typing Information Present.

This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

The following is the format for each bundle type:

2.8.1 Unused Entry

00h	CNT	TYPE

CNT = DB Number of entries.

This is the number of unused entries to skip.

TYPE = DB 0 (Unused Entry)

2.8.2 16-bit Entry

00h	CNT	TYPE	OBJECT
04h	FLAGS	OFFSET	
07h	...	...	

CNT = DB Number of entries.

This is the number of 16-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 1 (16-bit Entry)
OBJECT = DW Object number.

This is the object number for the entries in this bundle. If the object number is zero, then the entries in this bundle are absolute values.

FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.

2.8.3 286 Call Gate Entry

00h	CNT	TYPE	OBJECT	
04h	FLAGS	OFFSET		CALLGATE
09h	...	...		...

CNT = DB Number of entries.

This is the number of 286 call gate entries in this bundle. The flags, callgate, and offset value are repeated this number of times.

TYPE = DB 2 (286 Call Gate Entry)

The 286 Call Gate Entry Point type is needed by the loader only if ring 2 segments are to be supported. 286 Call Gate entries contain 2 extra bytes which are used by the loader to store an LDT callgate selector value.

OBJECT = DW Object number.

This is the object number for the entries in this bundle.

FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.

CALLGATE = DW Callgate selector.

The callgate selector is a reserved field used by the loader to store a call gate selector value for references to ring 2 entry points. When a ring 3 reference to a ring 2 entry point is made, the callgate selector with a zero offset is place in the relocation fixup address. The segment number and offset in segment is placed in the LDT callgate.

2.8.4 32-bit Entry

00h	CNT	TYPE	OBJECT
04h	FLAGS	OFFSET	
09h	...	...	

CNT = DB Number of entries.

This is the number of 32-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 3 (32-bit Entry)

The 32-bit Entry type will only be defined by the linker when the offset in the object can not be specified by a 16-bit offset.

OBJECT = DW Object number.

This is the object number for the entries in this bundle. If the object number is zero, then the entries in this bundle are absolute values.

FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter dword count mask.

OFFSET = DD Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.

2.8.5 Forwarder Entry

00h	CNT	TYPE	RESERVED	
04h	FLAGS	MOD ORD#		OFFSET / ORDNUM
09h	...	...		...

CNT = DB Number of entries.

This is the number of forwarder entries in this bundle. The FLAGS, MOD ORD#, and OFFSET/ORDNUM values are repeated this number of times.

TYPE = DB 4 (Forwarder Entry)
RESERVED = DW 0

This field is reserved for future use.

FLAGS = DB Forwarder flags.

These are the flags for this entry point. They have the following definition.

01h = Import by ordinal.
F7h = Reserved for future use; should be zero.

MOD ORD# = DW Module Ordinal Number

This is the index into the Import Module Name Table for this forwarder.

OFFSET / ORDNUM = DD Procedure Name Offset or Import Ordinal Number

If the FLAGS field indicates import by ordinal, then this field is the ordinal number into the Entry Table of the target module, otherwise this field is the offset into the Procedure Names Table of the target module.

A Forwarder entry (type = 4) is an entry point whose value is an imported reference. When a load time fixup occurs whose target is a forwarder, the loader obtains the address imported by the forwarder and uses that imported address to resolve the fixup.

A forwarder may refer to an entry point in another module which is itself a forwarder, so there can be a chain of forwarders. The loader will traverse the chain until it finds a non-forwarded entry point which terminates the chain, and use this to resolve the original fixup. Circular chains are detected by the loader and result in a load time error. A maximum of 1024 forwarders is allowed in a chain; more than this results in a load time error.

Forwarders are useful for merging and recombining API calls into different sets of libraries, while maintaining compatibility with applications. For example, if one wanted to combine MONCALLS, MOUCALLS, and VIOCALLS into a single libraries, one could provide entry points for the three libraries that are forwarders pointing to the common implementation.

2.9 Module Format Directives Table

The Module Format Directives Table is an optional table that allows additional options to be specified. It also allows for the extension of the linear EXE format by allowing additional tables of information to be added to the linear EXE module without affecting the format of the linear EXE header. Likewise, module format directives provide a place in the linear EXE module for

'temporary tables' of information, such as incremental linking information and statistic information gathered on the module. When there are no module format directives for a linear EXE module, the fields in the linear EXE header referencing the module format directives table are zero.

Each Module Format Directive Table entry has the following format:

Figure 3-13: Module Format Directive Table

00h	DIRECT #	DATA LEN	DATA OFFSET

DIRECT # = DW Directive number.

The directive number specifies the type of directive defined. This can be used to determine the format of the information in the directive data. The following directive numbers have been defined:

8001h = Verify Record Directive. (Verify record is a resident table.)
0002h = Language Information Directive. (This is a non-resident table.)
0003h = Co-Processor Required Support Table.
0004h = Thread State Initialization Directive.
0005h = C Set ++ Browse Information. Additional directives can be added as needed in the future, as long as they do not overlap previously defined directive numbers.

8000h = Resident Flag Mask.
Directive numbers with this bit set indicate that the directive data is in the resident area and will be kept resident in memory when the module is loaded.

DATA LEN = DW Directive data length.

This specifies the length in bytes of the directive data for this directive number.

DIRECTIVE OFFSET = DD Directive data offset.

This is the offset to the directive data for this directive number. It is relative to beginning of linear EXE header for a resident table, and relative to the beginning of the EXE file for non-resident tables.

2.9.1 Verify Record Directive Table

The Verify Record Directive Table is an optional table. It maintains a record of the pages in the EXE file that have been fixed up and written back to the original linear EXE module, along with the module dependencies used to perform these fixups. This table provides an efficient means for verifying the virtual addresses required for the fixed up pages when the module is loaded.

Each Verify Record entry has the following format:

Figure 3-14: Verify Record Table

00h	# OF ENTRY		
02h	MOD ORD #	VERSION	MOD # OBJ
08h	OBJECT #	BASE ADDR	VIRTUAL
0Eh	. . .	. . .	. . .

OF ENTRY = DW Number of module dependencies.

This field specifies how many entries there are in the verify record directive table. This is equal to the number of modules referenced by this module.

MOD ORD # = DW Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the referenced module.

VERSION = DW Module Version.

This is the version of the referenced module that the fixups were originally performed. This is used to insure the same version of the referenced module is loaded that was fixed up in this module and therefore the fixups are still correct. This requires the version number in a module to be incremented anytime the entry point offsets change.

MOD # OBJ = DW Module # of Object Entries.

This field is used to identify the number of object verify entries that follow for the referenced module.

OBJECT # = DW Object # in Module.

This field specifies the object number in the referenced module that is being verified.

BASE ADDR = DW Object load base address.

This is the address that the object was loaded at when the fixups were performed.

VIRTUAL = DW Object virtual address size.

This field specifies the total amount of virtual memory required for this object.

2.10 Per-Page Checksum

The Per-Page Checksum table provides space for a cryptographic checksum for each physical page

in the EXE file.

The checksum table is arranged such that the first entry in the table corresponds to the first logical page of code/data in the EXE file (usually a preload page) and the last entry corresponds to the last logical page in the EXE file (usually a iterated data page).

Figure 3-15: Per-Page Checksum

Logical Page #1	CHECKSUM
Logical Page #2	CHECKSUM
	. . .
Logical Page #n	CHECKSUM

CHECKSUM = DD Cryptographic checksum.

2.11 Fixup Page Table

The Fixup Page Table provides a simple mapping of a logical page number to an offset into the Fixup Record Table for that page.

This table is parallel to the Object Page Table, except that there is one additional entry in this table to indicate the end of the Fixup Record Table.

The format of each entry is:

Figure 3-16: Fixup Page Table

Logical Page #1	OFFSET FOR PAGE #1	
Logical Page #2	OFFSET FOR PAGE #2	
	. . .	
Logical Page #n	OFFSET FOR PAGE #n	
	OFF TO END OF FIXUP REC	This is equal to: Offset for page #n + Size of fixups for page #n

OFFSET FOR PAGE # = DD Offset for fixup record for this page.

This field specifies the offset, from the beginning of the fixup record table, to the first fixup record for this page.

OFF TO END OF FIXUP REC = DD Offset to the end of the fixup records.

This field specifies the offset following the last fixup record in the fixup record table. This is the last entry in the fixup page table.
The fixup records are kept in order by logical page in the fixup record table. This allows the end of each page's fixup records is defined by the offset for the next logical page's fixup records. This last entry provides support of this mechanism for the last page in the fixup page table.

2.12 Fixup Record Table

The Fixup Record Table contains entries for all fixups in the linear EXE module. The fixup records for a logical page are grouped together and kept in sorted order by logical page number. The fixups for each page are further sorted such that all external fixups and internal selector/pointer fixups come before internal non-selector/non-pointer fixups. This allows the loader to ignore internal fixups if the loader is able to load all objects at the addresses specified in the object table.

Each relocation record has the following format:

Figure 3-17: Fixup Record Table

00h	SRC	FLAGS	SRCOFF/CNT1	
03h/04h	TARGET DATA1			
	SRCOFF12	. . .		SRCOFFn2
	1These fields are variable size. 2These fields are optional.			

SRC = DB Source type.

The source type specifies the size and type of the fixup to be performed on the fixup source. The source type is defined as follows:

0Fh = Source mask.
00h = Byte fixup (8-bits).
01h = (undefined).
02h = 16-bit Selector fixup (16-bits).
03h = 16:16 Pointer fixup (32-bits).
04h = (undefined).
05h = 16-bit Offset fixup (16-bits).
06h = 16:32 Pointer fixup (48-bits).

07h = 32-bit Offset fixup (32-bits).
08h = 32-bit Self-relative offset fixup (32-bits).
10h = Fixup to Alias Flag.

When the 'Fixup to Alias' Flag is set, the source fixup refers to the 16:16 alias for the object. This is only valid for source types of 2, 3, and 6. For fixups such as this, the linker and loader will be required to perform additional checks such as ensuring that the target offset for this fixup is less than 64K.

20h = Source List Flag.

When the 'Source List' Flag is set, the SRCOFF field is compressed to a byte and contains the number of source offsets, and a list of source offsets follows the end of fixup record (after the optional additive value).

FLAGS = DB Target Flags.

The target flags specify how the target information is interpreted. The target flags are defined as follows:

03h = Fixup target type mask.
00h = Internal reference.
01h = Imported reference by ordinal.
02h = Imported reference by name.
03h = Internal reference via entry table.
04h = Additive Fixup Flag.

When set, an additive value trails the fixup record (before the optional source offset list).

08h = Internal Chaining Fixup Flag.

When set, this bit indicates that this fixup record is the beginning of a chain of fixups. The remaining fixups in the chain are contained within the page rather than as additional entries in the fixup record table. See § 2.12.5, "Internal Chaining Fixups" for additional details on this fixup type.
Setting this bit is only valid for fixups of source type 07h (32-bit Offset) and target types 00h and 03h (Internal references). This bit is only valid if the Source List Flag is not set.

10h = 32-bit Target Offset Flag.

When set, the target offset is 32-bits, otherwise it is 16-bits.

20h = 32-bit Additive Fixup Flag.

When set, the additive value is 32-bits, otherwise it is 16-bits.

40h = 16-bit Object Number/Module Ordinal Flag.

When set, the object number or module ordinal number is 16-bits, otherwise it is 8-bits.

80h = 8-bit Ordinal Flag.

When set, the ordinal number is 8-bits, otherwise it is 16-bits.

SRCOFF = DW/CNT = DB Source offset or source offset list count.

This field contains either an offset or a count depending on the Source List Flag. If the Source List Flag is set, a list of source offsets follows the additive field and this field contains the count of the entries in the source offset list. Otherwise, this is the single source offset for the fixup. Source offsets are relative to the beginning of the page where the fixup is to be made.

Note: For fixups that cross page boundaries, a separate fixup record is specified for each page. An offset is still used for the 2nd page but it now becomes a negative offset since the fixup originated on the preceding page. (For example, if only the last one byte of a 32-bit address is on the page to be fixed up, then the offset would have a value of -3.)

TARGET DATA = Target data for fixup.

The format of the TARGET DATA is dependent upon target flags.

SRCOFF1 - SRCOFFn = DW[] Source offset list.

This list is present if the Source List Flag is set in the Target Flags field. The number of entries in the source offset list is defined in the SRCOFF/CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.

2.12.1 Internal Fixup Record

Figure 3-18: Internal Fixup Record

00h	SRC	FLAGS	SRCOFF/CNT1	
03h/04h	OBJECT1		TRGOFF1,2	
	SRCOFF12	. . .		SRCOFFn2
	1These fields are variable size. 2These fields are optional.			

OBJECT = D[B|W] Target object number.

This field is an index into the current module's Object Table to specify the target Object. It is a Byte value when the '16-bit Object Number/Module Ordinal Flag' bit in the target flags field is clear and a Word value when the bit is set.

TRGOFF = D[W|D] Target offset.

This field is an offset into the specified target Object. It is not present when the Source Type specifies a 16-bit Selector fixup. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

2.12.2 Import by Ordinal Fixup Record

Figure 3-19: Import by Ordinal Fixup Record

00h	SRC	FLAGS	SRCOFF/CNT1	
03h/04h	MOD ORD#1		IMPORT ORD1	ADDITIVE1,2
	SRCOFF12		. . .	SRCOFFn2
	1These fields are variable size. 2These fields are optional.			

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

IMPORT ORD = D[B|W|D] Imported ordinal number.

This is the imported procedure's ordinal number. It is a Byte value when the '8-bit Ordinal' bit in the target flags field is set. Otherwise it is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).
This value is added to the address derived from the target entry point. This field is a

Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

2.12.3 Import by Name Fixup Record

Figure 3-20: Import by Name Fixup Record

00h	SRC	FLAGS	SRCOFF/CNT1			
03h/04h	MOD ORD#1		PROCEDURE NAME OFFSET1		ADDITIVE1,2	
	SRCOFF12		. . .	SRCOFFn2		
	1These fields are variable size. 2These fields are optional.					

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

PROCEDURE NAME OFFSET = D[W|D] Offset into the Import Procedure Name Table.

This field is an offset into the Import Procedure Name Table. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).
This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

2.12.4 Internal Entry Table Fixup Record

Figure 3-21: Internal Entry Table Fixup Record

00h	SRC	FLAGS	SRCOFF/CNT1	
03h/04h	ORD #1		ADDITIVE1,2	
	SRCOFF12	. . .		SRCOFFn2
	1These fields are variable size. 2These fields are optional.			

ENTRY # = D[B|W] Ordinal index into the Entry Table.

This field is an index into the current module's Entry Table to specify the target Object and offset. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).
This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

2.12.5 Internal Chaining Fixups

Internal chaining fixups are 32-bit offset fixups (source type 07h) to internal references (target types 00h and 03h) where the first fixup in the chain is a record in the Fixup Record Table and the remaining fixups are located in the page referenced by the first fixup rather than as records in the Fixup Record Table. The chain is a linked-list with each fixup pointing to the next fixup in the chain until the end of the chain is reached. All fixups in the chain must fixup source offsets within the same page. All fixups in the chain must reference targets within the same memory object. All target references within a chain are also restricted to be within a 1 MB virtual memory range. Of course multiple chains can be used to meet the restrictions on an individual chain.

The first fixup in the chain is contained in a record in the Fixup Record Table with the 'Internal Chaining Fixup Flag' bit in the target flags field set. This fixup is otherwise normal except that the 32-bit source location contains information about this fixup and the next fixup in the chain.

Before applying an internal chaining fixup, the 32-bit value at the source location should be read. The first 12-bits of the value contain the source offset, within the page, for the next fixup in the chain and the remaining 20-bits contain the target offset of the current fixup.

Figure 3-22: Internal Chaining Fixup Source Location

< 12 bits >	< 20 bits >
SRCOFF	TRGOFF

SRCOFF = Source offset.

This field contains the source offset, from the start of the page, to the *next* fixup in the chain. The final fixup in the chain is identified by the value of 0xFFF indicating no further fixups in the chain.

TRGOFF = Target offset.

This field contains the target offset from a base target address for the *current* fixup. A base target address for the all the fixups in the chain is computed by subtracting the TRGOFF value, found in the source location of the first fixup, from the target address computed for the first fixup, using the normal technique for the record in the Fixup Record Table. This base target address is then added to each TRGOFF in the fixup chain to compute the proper target address for the individual fixup.
Since TRGOFF is limited to 20-bits, there is a 1 MB virtual memory range for the targets of all the fixups in a chain. The range is base target address to base target address + (1MB - 1).

The following pseudocode illustrates how the chain of fixups work.

Figure 3-23: Internal Chaining Fixup Pseudocode

```
source_page = address of source page containing fixups
source_offset = offset in page of fixup (from Fixup Record Table record)
target_address = address of target reference (from Fixup Record Table record)

if (fixup is an Internal Chaining Fixup)
{
        fixupinfo = *(source_page + source_offset)
        base_target_address = target_address - (fixupinfo & 0xFFFFF)
        do
        {
                fixupinfo = *(source_page + source_offset)
                *(source_page + source_offset) =
                                base_target_address + (fixupinfo & 0xFFFFF)
                source_offset = (fixupinfo >> 20) & 0xFFF)
        }
        while (source_offset != 0xFFF)
}
else
{
        *(source_page + source_offset) = target_address
}
```

2.13 Import Module Name Table

The import module name table defines the module name strings imported through dynamic link

references. These strings are referenced through the imported relocation fixups.

To determine the length of the import module name table subtract the import module name table offset from the import procedure name table offset. These values are located in the linear EXE header. The end of the import module name table is not terminated by a special character, it is followed directly by the import procedure name table.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

Figure 3-24: Import Module Name Table

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 255 characters.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

2.14 Import Procedure Name Table

The import procedure name table defines the procedure name strings imported by this module through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import procedure name table add the fixup section size to the fixup page table offset, this computes the offset to the end of the fixup section, then subtract the import procedure name table offset. These values are located in the linear EXE header. The import procedure name table is followed by the data pages section.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

Figure 3-25: Import Procedure Name Table

00h	LEN	ASCII STRING ...

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 255 characters.
The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Note: The first entry in the import procedure name table must be a null entry. That is, the LEN field should be zero followed an empty ASCII STRING (no bytes).

2.15 Preload Pages

The Preload Pages section is an optional section in the linear EXE module that coalesces a 'preload page set' into a contiguous section. The preload page set can be defined as the set of first used pages in the module. The preload page set can be specified by the application developer or can be derived by a tool that analyzes the programs memory usage while it is running. By grouping the preload page set together, the preload pages can be read from the linear EXE module with one disk read.

The structure of the preload pages is no different than if they were demand loaded. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded.

All pages begin on a PAGE OFFSET SHIFT boundary from the base of the preload page section, as specified in the linear EXE header. The pages are ordered by logical page number within this section.

Note: OS/2 2.x does not respect the preload of pages as specified in the executable file for performance reasons.

2.16 Demand Load Pages

The Demand Loaded Pages section contains all the non-iterated pages for a linear EXE module that are not preloaded. When required, the whole page is loaded into memory from the module. The

characteristics of each of these pages is specified in the Object Page Table. Every page begins on a PAGE OFFSET SHIFT boundary aligned offset from the demand loaded pages base specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded. The pages are ordered by logical page number within this section.

2.17 Iterated Data Pages

The Iterated Data Pages section contains all the pages for a linear EXE module that are iterated. When required, the set of iteration records are loaded into memory from the module and expanded to reconstitute the page. Every set of iteration records begins on a PAGE OFFSET SHIFT offset from the OBJECT ITER PAGES OFF specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. The pages are ordered by logical page number within this section.

This record structure is used to describe the iterated data for an object on a per-page basis.

Figure 3-26: Object Iterated Data Record
(Iteration Record)

00h	#ITERATIONS	DATA LENGTH	
04h	DATA BYTES	...	...

#ITERATIONS = DW Number of iterations.

This specifies the number of times that the data is replicated.

DATA LENGTH = DW The size of the data pattern in bytes.

This specifies the number of bytes of data of which the pattern consists. The maximum size is one half of the PAGE SIZE (given in the module header). If a pattern exceeds this value then the data page will not be condensed into iterated data.

DATA = DB * DATA LENGTH The Data pattern to be replicated.

The next iteration record will immediately follow the last byte of the pattern. The offset of the next iteration record is easily calculated from the offset of this record by adding the DATA LENGTH field and the sizes of the #ITERATIONS and DATA LENGTH fields.

2.18 Debug Information

The debug information is defined by the debugger and is not controlled by the linear EXE format or linker. The only data defined by the linear EXE format relative to the debug information is it's offset in the EXE file and length in bytes as defined in the linear EXE header.

To support multiple debuggers the first word of the debug information is a type field which determines the format of the debug information.

Figure 3-27: Debug Information

00h	01h	02h	03h	04h
'N'	'B'	'0'	n	DEBUGGER DATA . . . .

TYPE = DB DUP 4 Format type.

This defines the type of debugger data that exists in the remainder of the debug information. The signature consists of a string of four (4) ASCII characters: "NB0" followed by the ASCII representation for 'n'. The values for 'n' are defined as follows. These format types are defined.

00h = 32-bit CodeView debugger format.
01h = AIX debugger format.
02h = 16-bit CodeView debugger format.
04h = 32-bit OS/2 PM debugger (IBM) format.
DEBUGGER DATA = Debugger specific data.

The format of the debugger data is defined by the debugger that is being used. The values defined for the type field are not enforced by the system. It is the responsibility of the linker or debugging tools to follow the convention for the type field that is defined here.

---

[1] Windows is a Registered Trademark of Microsoft Corp.
