.EXE Executable-File Header Format (3.1)

An executable (.EXE) file for the Microsoft Windows operating system contains a combination of code and data or a combination of code, data, and resources. The executable file also contains two headers: an MS-DOS header and a Windows header. The next two sections describe these headers; the third section describes the code and data contained in a Windows executable file.

MS-DOS Header

The MS-DOS (old-style) executable-file header contains four distinct parts: a collection of header information (such as the signature word, the file size, and so on), a reserved section, a pointer to a Windows header (if one exists), and a stub program. The following illustration shows the MS-DOS executable-file header:
If the word value at offset 18h is 40h or greater, the word value at 3Ch is typically an offset to a Windows header. Applications must verify this for each executable-file header being tested, because a few applications have a different header style.
MS-DOS uses the stub program to display a message if Windows has not been loaded when the user attempts to run a program.
For more information about the MS-DOS executable-file header, see the Microsoft MS-DOS Programmer's Reference (Redmond, Washington: Microsoft Press, 1991).

Windows Header

The Windows (new-style) executable-file header contains information that the loader requires for segmented executable files. This information includes the linker version number, data specified by the linker, data specified by the resource compiler, tables of segment data, tables of resource data, and so on. The following illustration shows the Windows executable-file header:
The following sections describe the entries in the Windows executable-file header.

Information Block

The information block in the Windows header contains the linker version number, the lengths of various tables that further describe the executable file, the offsets from the beginning of the header to the beginning of these tables, the heap and stack sizes, and so on. The following list summarizes the contents of the header information block (the locations are relative to the beginning of the block):

Location   Description

00h   Specifies the signature word. The low byte contains "N" (4Eh) and the high byte contains "E" (45h).
02h   Specifies the linker version number.
03h   Specifies the linker revision number.
04h   Specifies the offset to the entry table (relative to the beginning of the header).
06h   Specifies the length of the entry table, in bytes.
08h   Reserved.
0Ch   Specifies flags that describe the contents of the executable file. This value can be one or more of the following bits:

Bit   Meaning

0   The linker sets this bit if the executable-file format is SINGLEDATA. An executable file with this format contains one data segment. This bit is set if the file is a dynamic-link library (DLL).
1   The linker sets this bit if the executable-file format is MULTIPLEDATA. An executable file with this format contains multiple data segments. This bit is set if the file is a Windows application.

If neither bit 0 nor bit 1 is set, the executable-file format is NOAUTODATA. An executable file with this format does not contain an automatic data segment.

2 Reserved.
3 Reserved.
8 Reserved.
9 Reserved.
11 If this bit is set, the first segment in the executable file contains code that loads the application.
13 If this bit is set, the linker detects errors at link time but still creates an executable file.
14 Reserved.
15 If this bit is set, the executable file is a library module.

If bit 15 is set, the CS:IP registers point to an initialization procedure called with the value in the AX register equal to the module handle. The initialization procedure must execute a far return to the caller. If the procedure is successful, the value in AX is nonzero. Otherwise, the value in AX is zero.
The value in the DS register is set to the library's data segment if SINGLEDATA is set. Otherwise, DS is set to the data segment of the application that loads the library.

0Eh Specifies the automatic data segment number. (0Eh is zero if the SINGLEDATA and MULTIPLEDATA bits are cleared.)
10h Specifies the initial size, in bytes, of the local heap. This value is zero if there is no local allocation.
12h Specifies the initial size, in bytes, of the stack. This value is zero if the SS register value does not equal the DS register value.
14h Specifies the segment:offset value of CS:IP.
18h Specifies the segment:offset value of SS:SP.

The value specified in SS is an index to the module's segment table. The first entry in the segment table corresponds to segment number 1.
If SS addresses the automatic data segment and SP is zero, SP is set to the address obtained by adding the size of the automatic data segment to the size of the stack.

1Ch Specifies the number of entries in the segment table.
1Eh Specifies the number of entries in the module-reference table.
20h Specifies the number of bytes in the nonresident-name table.
22h Specifies a relative offset from the beginning of the Windows header to the beginning of the segment table.
24h Specifies a relative offset from the beginning of the Windows header to the beginning of the resource table.
26h Specifies a relative offset from the beginning of the Windows header to the beginning of the resident-name table.
28h Specifies a relative offset from the beginning of the Windows header to the beginning of the module-reference table.
2Ah Specifies a relative offset from the beginning of the Windows header to the beginning of the imported-name table.
2Ch Specifies a relative offset from the beginning of the file to the beginning of the nonresident-name table.

30h Specifies the number of movable entry points.
32h Specifies a shift count that is used to align the logical sector. This count is log2 of the segment sector size. It is typically 4, although the default count is 9. (This value corresponds to the /alignment [/a] linker switch. When the linker command line contains /a:16, the shift count is 4. When the linker command line contains /a:512, the shift count is 9.)
34h Specifies the number of resource segments.
36h Specifies the target operating system, depending on which bits are set:

Bit Meaning

0 Operating system format is unknown.
1 Reserved.

2 Operating system is Microsoft Windows.
3 Reserved.
4 Reserved.

37h Specifies additional information about the executable file. It can be one or more of the following values:

Bit Meaning

1 If this bit is set, the executable file contains a Windows 2.x application that runs in version 3.x protected mode.
2 If this bit is set, the executable file contains a Windows 2.x application that supports proportional fonts.
3 If this bit is set, the executable file contains a fast-load area.

38h Specifies the offset, in sectors, to the beginning of the fast-load area. (Only Windows uses this value.)
3Ah Specifies the length, in sectors, of the fast-load area. (Only Windows uses this value.)
3Ch Reserved.
3Eh Specifies the expected version number for Windows. (Only Windows uses this value.)

Segment Table

The segment table contains information that describes each segment in an executable file. This information includes the segment length, segment type, and segment-relocation data. The following list summarizes the values found in the segment table (the locations are relative to the beginning of each entry):

Location Description

00h Specifies the offset, in sectors, to the segment data (relative to the beginning of the file). A value of zero means no data exists.
02h Specifies the length, in bytes, of the segment, in the file. A value of zero indicates that the segment length is 64K, unless the selector offset is also zero.
04h Specifies flags that describe the contents of the executable file. This value can be one or more of the following:

Bit Meaning

0 If this bit is set, the segment is a data segment. Otherwise, the segment is a code segment.
1 If this bit is set, the loader has allocated memory for the segment.
2 If this bit is set, the segment is loaded.
3 Reserved.
4 If this bit is set, the segment type is MOVABLE. Otherwise, the segment type is FIXED.
5 If this bit is set, the segment type is PURE or SHAREABLE. Otherwise, the segment type is IMPURE or NONSHAREABLE.
6 If this bit is set, the segment type is PRELOAD. Otherwise, the segment type is LOADONCALL.
7 If this bit is set and the segment is a code segment, the segment type is EXECUTEONLY. If this bit is set and the segment is a data segment, the segment type is READONLY.

8 If this bit is set, the segment contains relocation data.
9 Reserved.
10 Reserved.
11 Reserved.
12 If this bit is set, the segment is discardable.
13 Reserved.

14	Reserved.
15	Reserved.

06h Specifies the minimum allocation size of the segment, in bytes. A value of zero indicates that the minimum allocation size is 64K.

Resource Table

The resource table describes and identifies the location of each resource in the executable file. The table has the following form:

```
WORD     rscAlignShift;
TYPEINFO rscTypes[];
WORD     rscEndTypes;
BYTE     rscResourceNames[];
BYTE     rscEndNames;
```

Following are the members in the resource table:

rscAlignShift Specifies the alignment shift count for resource data. When the shift count is used as an exponent of 2, the resulting value specifies the factor, in bytes, for computing the location of a resource in the executable file.

rscTypes Specifies an array of TYPEINFO structures containing information about resource types. There must be one TYPEINFO structure for each type of resource in the executable file.

rscEndTypes Specifies the end of the resource type definitions. This member must be zero.

rscResourceNames Specifies the names (if any) associated with the resources in this table. Each name is stored as consecutive bytes; the first byte specifies the number of characters in the name.

rscEndNames Specifies the end of the resource names and the end of the resource table. This member must be zero.

Type Information

The TYPEINFO structure has the following form:

```
typedef struct _TYPEINFO {
    WORD        rtTypeID;
    WORD        rtResourceCount;
    DWORD       rtReserved;
    NAMEINFO    rtNameInfo[];
} TYPEINFO;
```

Following are the members in the TYPEINFO structure:

rtTypeID Specifies the type identifier of the resource. This integer value is either a resource-type value or an offset to a resource-type name. If the high bit in this member is set (0x8000), the value is one of the following resource-type values:

Value	Resource type

RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap
RT_CURSOR	Cursor
RT_DIALOG	Dialog box
RT_FONT	Font component
RT_FONTDIR	Font directory
RT_GROUP_CURSOR	Cursor directory
RT_GROUP_ICON	Icon directory
RT_ICON	Icon
RT_MENU	Menu
RT_RCDATA	Resource data
RT_STRING	String table

If the high bit of the value in this member is not set, the value represents an offset, in bytes relative to the beginning of the resource table, to a name in the rscResourceNames member.

rtResourceCount Specifies the number of resources of this type in the executable file.
rtReserved Reserved.
rtNameInfo Specifies an array of NAMEINFO structures containing information about individual resources. The rtResourceCount member specifies the number of structures in the array.

Name Information

The NAMEINFO structure has the following form:

```
typedef struct _NAMEINFO {
    WORD rnOffset;
    WORD rnLength;
    WORD rnFlags;
    WORD rnID;
    WORD rnHandle;
    WORD rnUsage;
} NAMEINFO;
```

Following are the members in the NAMEINFO structure:

rnOffset Specifies an offset to the contents of the resource data (relative to the beginning of the file). The offset is in terms of alignment units specified by the rscAlignShift member at the beginning of the resource table.
rnLength Specifies the resource length, in bytes.
rnFlags Specifies whether the resource is fixed, preloaded, or shareable. This member can be one or more of the following values:

Value	Meaning
0x0010	Resource is movable (MOVEABLE). Otherwise, it is fixed.
0x0020	Resource can be shared (PURE).
0x0040	Resource is preloaded (PRELOAD). Otherwise, it is loaded on demand.

rnID Specifies or points to the resource identifier. If the identifier is an integer, the high bit is set (8000h). Otherwise, it is an offset to a resource string, relative to the beginning of the resource table.
rnHandle Reserved.
rnUsage Reserved.

Resident-Name Table

The resident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are resident in system memory and are never discarded. The resident-name strings are case-sensitive and are not null-terminated. The following list summarizes the values found in the resident-name table (the locations are relative to the beginning of each entry):

Location Description

00h Specifies the length of a string. If there are no more strings in the table, this value is zero.
01h - xxh Specifies the resident-name text. This string is case-sensitive and is not null-terminated.
xxh + 01h Specifies an ordinal number that identifies the string. This number is an index into the entry table.

The first string in the resident-name table is the module name.

Module-Reference Table

The module-reference table contains offsets for module names stored in the imported-name table. Each entry in this table is 2 bytes long.

Imported-Name Table

The imported-name table contains the names of modules that the executable file imports. Each entry contains two parts: a single byte that specifies the length of the string and the string itself. The strings in this table are not null-terminated.

Entry Table

The entry table contains bundles of entry points from the executable file (the linker generates each bundle). The numbering system for these ordinal values is 1-based--that is, the ordinal value corresponding to the first entry point is 1.
The linker generates the densest possible bundles under the restriction that it cannot reorder the entry points. This restriction is necessary because other executable files may refer to entry points within a given bundle by their ordinal values.
The entry-table data is organized by bundle, each of which begins with a 2-byte header. The first byte of the header specifies the number of entries in the bundle (a value of 00h designates the end of the table). The second byte specifies whether the corresponding segment is movable or fixed. If the value in this byte is 0FFh, the segment is movable. If the value in this byte is 0FEh, the entry does not refer to a segment but refers, instead, to a constant defined within the module. If the value in this byte is neither 0FFh nor 0FEh, it is a segment index.

For movable segments, each entry consists of 6 bytes and has the following form:

Location Description

00h Specifies a byte value. This value can be a combination of the following bits:

Bit(s) Meaning

0 If this bit is set, the entry is exported.

1 If this bit is set, the segment uses a global (shared) data segment.

3-7 If the executable file contains code that performs ring transitions, these bits specify the number of words that compose the stack. At the time of the ring transition, these words must be copied from one ring to the other.

01h Specifies an int 3fh instruction.

03h Specifies the segment number.

04h Specifies the segment offset.

For fixed segments, each entry consists of 3 bytes and has the following form:

Location Description

00h Specifies a byte value. This value can be a combination of the following bits:

Bit(s) Meaning

0 If this bit is set, the entry is exported.

1 If this bit is set, the entry uses a global (shared) data segment. (This may be set only for SINGLEDATA library modules.)

3-7 If the executable file contains code that performs ring transitions, these bits specify the number of words that compose the stack. At the time of the ring transition, these words must be copied from one ring to the other.

01h Specifies an offset.

Nonresident-Name Table

The nonresident-name table contains strings that identify exported functions in the executable file. As the name implies, these strings are not always resident in system memory and are discardable. The nonresident-name strings are case-sensitive; they are not null-terminated. The following list summarizes the values found in the nonresident-name table (the specified locations are relative to the beginning of each entry):

Location Description

00h Specifies the length, in bytes, of a string. If this byte is 00h, there are no more strings in the table.

01h - xxh Specifies the nonresident-name text. This string is case-sensitive and is not null-terminated.

xx + 01h Specifies an ordinal number that is an index to the entry table.

The first name that appears in the nonresident-name table is the module description string (which was specified in the module-definition file).

Code Segments and Relocation Data

Code and data segments follow the Windows header. Some of the code segments may contain calls to functions in other segments and may, therefore, require relocation data to resolve those references. This relocation data is stored in a relocation table that appears immediately after the code or data in the segment. The first 2 bytes in this table specify the number of relocation items the table contains. A relocation item is a collection of bytes specifying the following information:

Address type (segment only, offset only, segment and offset)

Relocation type (internal reference, imported ordinal, imported name)

Segment number or ordinal identifier (for internal references)

Reference-table index or function ordinal number (for imported ordinals)

Reference-table index or name-table offset (for imported names)

Each relocation item contains 8 bytes of data, the first byte of which specifies one of the following relocation-address types:

Value	Meaning
0	Low byte at the specified offset
2	16-bit selector
3	32-bit pointer
5	16-bit offset
11	48-bit pointer
13	32-bit offset

The second byte specifies one of the following relocation types:

Value	Meaning
0	Internal reference
1	Imported ordinal
2	Imported name
3	OSFIXUP

The third and fourth bytes specify the offset of the relocation item within the segment.
If the relocation type is imported ordinal, the fifth and sixth bytes specify an index to a module's reference table and the seventh and eighth bytes specify a function ordinal value.
If the relocation type is imported name, the fifth and sixth bytes specify an index to a module's reference table and the seventh and eighth bytes specify an offset to an imported-name table.
If the relocation type is internal reference and the segment is fixed, the fifth byte specifies the segment number, the sixth byte is zero, and the seventh and eighth bytes specify an offset to the segment. If the relocation type is internal reference and the segment is movable, the fifth byte specifies 0FFh, the sixth byte is zero; and the seventh and eighth bytes specify an ordinal value found in the segment's entry table.