Win32 Binary resource formats.

0. Preface.

This document was edited and released by Microsoft Developer Support. It describes the binary format of resources in Win32. The information is provided at this point because we feel it would make the work of application development easier. Unfortunately, the information in this document may change before the final release of Windows/NT. Microsoft is NOT committing to stay with these formats by releasing this document. Questions or followups for any of the information presented here should be posted to Compuserve MSWIN32 forum, section 4. -Steve Firebaugh

1. Overview

This document details the structure of the resource binary file (.res) format for Windows NT and DOS Windows 32. The structure is similar to the existing Windows 16 (Win 3.0/3.1) structure, but supports essential new features such as UNICODE strings, version headers, and DWORD alignment. To support these changes, the file written by the resource compiler must be changed from Windows 16.

1.1 Comparison between Windows 3.0/3.1 and Windows 32

The Windows 16 resource file is a file containing one or more binary resources. Each resource is preceded by a variable length structure containing: Type, Name, Flags, Size. The Type and Name fields are either a string identifying the Type or Name, or a WORD value specifying the ordinal identity of the resource. The Flags field specifies to the system how to load into memory the resource, and the size specifies the size in bytes of the resource. The size, therefore, points to the next resource in the file.

The Windows 32 (both NT and DOS) resource file retains this structure, while expanding the header information with several additional values. It also adds a few fields to some of the pre-defined resources (Menu and Dialog, for instance), aligns all fields within the predefined resources on WORD or DWORD boundaries, and adds UNICODE (16-bit character) support to the data structures.

One additional difference in resource files for Windows 32 is worth noting. This does not directly affect the structure of the resource file, but is rather a difference in how resource files are handled and incorporated into an executable image (dll or exe). Windows NT uses Coff format objects. Because of this, and the fact that the Windows 32 exe format is much different than the Windows 16 format, the PDK provides a utility named CVTRES that converts a resource file into a Coff object. The linker then incorporates this object directly into the resulting executable image. No provision is made (as in Windows 16) for running the second pass of the resource compiler multiple times to update the resources: relinking the image is required.

However, the Windows 32 API provides a set of APIs that allow a program to enumerate all resources within an executable image, and update individual resources in the image.

1.2 Strings in UNICODE

All strings in a resource file are now stored in UNICODE format. In this format, all characters are represented by a 16-bit (WORD) value. The first 256 characters are identical to the 256 characters in the Windows ANSI character set (although the characters are represented by 16 bits each rather than 8bits). This means that they are terminated with a UNICODE_NULL symbol rather than a single NULL. The resource compiler translates all normal ASCII strings into UNICODE by calling the **MultiByteToWideChar** function provided by the Windows API. All escaped characters are stored directly, and are assumed to be valid UNICODE characters for the resource. If these strings are read in later by an application as ASCII (for instance, by calling the LoadString api), they will be converted back from UNICODE to ASCII transparently by the loader.

The only exception to the rule is strings in RCDATA statements. These psuedo-strings are not real strings, but merely a convenient notation for a collection of bytes. Users may overlay a structure over the data from an RCDATA statement, and expect certain data to be at certain offsets. If a psuedo-string gets automatically changed into a UNICODE string, it would inadvertently change the offsets of things in the structure and break those applications. Hence, these psuedo-strings must be left as ASCII bytes. To specify a UNICODE string inside an RCDATA statement, the user should use the explicit L-quoted string.

1.3 DWORD Alignment

To make resource binary files easier to read under Windows 32, all objects within the file are to be DWORD aligned. This includes headers, as well as data entries. This does not usually entail changes in the order of the fields of resource data structures, but does entail the need for some padding between fields.

The single exception to this rule is the font and fontdir structures. The reason for the exception is that these two structures are copied directly from other files, and are not used by RC.

2. General information

The resource file is created by the resource compiler (RC) while parsing the resource script file (.RC) and including any other resource data files (eg. .ICO, .CUR, .BMP, .FNT). The resource file contains all information necessary to build the resource table in the executable image. The main purpose of the resource file is to speed the edit-compile-link cycle by not always forcing resources to be recompiled.

There are currently about a dozen pre-defined resource types. These include Menus, Dialogs, Accelerators, Strings, Icons, Cursors, Bitmaps, Fonts and Version. These resources are used by the Windows system to define the appearance of the application window. The resource script allows the application writer to represent these features in an easily editable form. Other type ranges are reserved for use by the application for application-specific data. No attempt is made by the resource compiler to modify this user-defined data from 16-bit to 32-bit format.

The executable image file for Windows 32 is not a Segmented image. In the 16-bit executable image file, each resource was placed into a separate segment in the image file. The Windows 32 image file places all resources into a single Object or section. The Windows 32 image file also provides a binary-sorted resource table that allows fast lookup of a particular resource, rather than a table that must be searched linearly, as in the 16-bit image file. Because this Windows 32 image file format is more complex that the Windows 16 format, making it harder to update directly, the Windows 32 API provides methods of modifying the resource data directly.

The CVTRES conversion utility that converts the resource file into a coff object creates the resource table. This table contains two or three directories, indexed by Type, Name and Language, in that order. The Language directory is optional. The Type and Name directories consist of two parts: the part dedicated to resources whose types or names are represented by strings, and those represented by ordinal WORD values. Because the use of strings as resource type and name identifiers takes considerably more room than ordinal identifiers, Microsoft recommends that they not be used.

Note that, as all strings in a resource file (including the strings used to identify the type and name of resources) are UNICODE, the corresponding strings in the program that are being passed to LoadBitmap, etc., must also be UNICODE (only if the application is using the UNICODE set of apis rather than the ASCII set). This is facilitated by use of the TEXT macro provided in winnt.h.

The third level, language, provides the capability for the application developer to ship a single executable image that supports more than one language. For instance, one image providing French, French-Canadian and French-Belgium could be easily shipped in one image file. An application could also be shipped with support for all languages supported by the UNICODE standard, although this would probably make the image very large.

2.1 New statements

Several new statements have been added that the Windows 32 resource compiler processes.

2.1.1 New Button statements.

These statements allow the application developer the same freedom of expression that the PUSHBUTTON, DEFPUSHBUTTON, etc., statements do, and correspond to the appropriate button style.

They all have syntax identical to that of PUSHBUTTON, etc.

2.1.1.1 AUTO3STATE

Allows declaration of an AUTO3STATE button.

2.1.1.2 AUTOCHECKBOX

Allows declaration of an AUTOCHECKBOX button.

2.1.1.3 AUTORADIOBUTTON

Allows declaration of an AUTORADIOBUTTON button.

2.1.1.5 PUSHBOX

Allows declaration of a PUSHBOX button.

2.1.1.6 STATE3

Allows declaration of a 3STATE button (the 3 is at the end for syntax purposes).

2.1.1.7 USERBUTTON

Allows declaration of a USERBUTTON user-defined button.

2.1.2 EXSTYLE statement

This statement allows the application developer to designate one of the extended (WS_EX_xxx) styles for a dialog or control window. There are three methods, depending upon what is needed.

It may be placed just below the DIALOG statement to apply to the dialog window (like the CAPTION or STYLE statements).

```
EXSTYLE <flags>
```

It may be placed on the DIALOG statement with the memory flags.

```
FOOBAR DIALOG [MemFlags...] [EXSTYLE=<flags>] x, y, dx, dy
```

It may also be placed on the individual CONTROL, PUSHBUTTON, LTEXT, etc. statements at the end of the statement.

```
AUTOCHECKBOX "autocheckbox", id, x, y, dx, dy [styleflags][exstyleflags]
```

2.1.3 CHARACTERISTICS statement

This statement allows the application developer to specify information about the resource that may be of use to tools that read and write resource files. It has no significance to the system and is not stored in the image file.

CHARACTERISTICS <user defined DWORD value>

2.1.4 VERSION statement

This statement is intended to allow the application to specify a version number of the resource in the resource file (for those tools that read and write resource files.) It has no significance to the system and is not stored in the image file.

VERSION <user defined DWORD value>

2.1.5 LANGUAGE statement

The LANGUAGE statement is used to specify the language the resource file, or section of resource file, is written in. It may be placed anywhere within the resource script file that a single-line statement (such as ICON, CURSOR, BITMAP) may be placed. The scope of the language specified by a LANGUAGE statement is from that point in the script file to the next LANGUAGE statement, or the end of the file.

LANGUAGE <majornumber>,<minornumber>

where <majornumber> represents the language id, and <minornumber> the sub-language identifiers. The values specified in winnls.h should be used.

The LANGUAGE statement may also be placed before the BEGIN statement for MENU, DIALOG, STRINGTABLE, ACCELERATOR and RCDATA resources, along with other optional statements like CAPTION, STYLE, etc. If the statement is placed here, it's scope is limited to the resource being defined.

2.1.6 MESSAGETABLE statement

The MESSAGETABLE statement is used to include a message table. A message table is a special-purpose string table that is used to contain error or informational messages, and may contain formatting information. The format is:

<nameid> MESSAGETABLE <filename>

3. Resource Header Format

The general format of the entire file is simply a number of resource file entries concatenated together. Each resource contains the information about a single resource, such as a dialog or a string table.

Each entry consists of a resource header followed by the data for that resource. A resource header (which is DWORD aligned) is composed of four elements: two dwords containing the size of the header and the size of the resource data, the resource type, the resource name, and additional resource information. The data for the resource follows the resource header and is specific to each particular type of resource.

3.1 DataSize

This field gives the size of the data that follows the header, not including any file padding between this resource and any resource that follows this resource in the resource file.

3.2 HeaderSize

The HeaderSize field gives the size of the resource header structure that follows.

3.3 Type

The type field can either be a number or a double-null-terminated UNICODE string specifying the name of the type. This variable kind of type is known as a `Name or Ordinal' field, and is used in most places in a resource file where an ID may appear.

The first WORD of a Name or Ordinal field identifies whether the field is a number or a string. If the first WORD is 0xffff (an invalid UNICODE character), then the following WORD of information is the type number. Otherwise, the field is specified by a UNICODE string.

If the type field is a number, then the number specifies a standard or user-defined resource type. All standard Windows resource types have been assigned numbers, which are listed below. This list is taken from the header file used to make RC and contains the type number of the various resource types:

```
/* Predefined resource types */

#define         RT_NEWRESOURCE          0x2000
#define         RT_ERROR                0x7fff
#define         RT_CURSOR               1
#define         RT_BITMAP               2
#define         RT_ICON                 3
#define         RT_MENU                 4
#define         RT_DIALOG               5
#define         RT_STRING               6
#define         RT_FONTDIR              7
#define         RT_FONT                 8
#define         RT_ACCELERATORS         9
#define         RT_RCDATA               10
#define         RT_MESSAGETABLE         11
#define         RT_GROUP_CURSOR         12
#define         RT_GROUP_ICON           14
#define         RT_VERSION              16
#define         RT_NEWBITMAP            (RT_BITMAP|RT_NEWRESOURCE)
#define         RT_NEWMENU              (RT_MENU|RT_NEWRESOURCE)
#define         RT_NEWDIALOG            (RT_DIALOG|RT_NEWRESOURCE)
```

If the type field is a string, then the type is a user-defined type.

3.4 Names

A name identifies the particular resource. A name (like a type) may be a number or a string, and they are distinguished in the same way as numbers and strings are distinguished in the type field.

Note that no padding (for DWORD alignment) is needed between the Type and Name fields, as they contain only WORD data and hence the Name field will always be properly aligned. However, there may need to be a WORD of padding after the Name field to align the rest of the header on DWORD boundaries.

3.5 Additional Header Information

The additional information contains more information about the particular resource data, including size and language ID. The structure of the Header, plus it's additional information is as follows:

```
struct tagResource {
      DWORD       DataSize;                    // size of data without header
      DWORD       HeaderSize;                  // Length of the additional header
      [Ordinal or name TYPE]                   // type identifier, id or string
      [Ordinal or name NAME]                   // name identifier, id or string
      DWORD       DataVersion;                 // predefined resource data version
      WORD        MemoryFlags;                 // state of the resource
      WORD        LanguageId;                  // UNICODE support for NLS
      DWORD       Version;                     // Version of the resource data
      DWORD       Characteristics;             // Characteristics of the data
      } ;
```

The additional information structure will always begin on a DWORD boundary within the resource file, which may require adding some padding in between the name field and the ResAdditional structure.

3.5.1 DataVersion

The version determines which version of the data within the resource data that follows. This may be used in the future to allow additional information to be entered into the predefined formats.

3.5.2 MemoryFlags

The field wMemoryFlags contains flags telling the state of a given resource. These attributes are given to a given resource by modifiers in the .RC script. The script identifiers inject the following flag values:

```
#define        MOVEABLE           0x0010
#define        FIXED              ~MOVEABLE
#define        PURE               0x0020
#define        IMPURE             ~PURE
#define        PRELOAD            0x0040
#define        LOADONCALL         ~PRELOAD
#define        DISCARDABLE        0x1000
```

The resource compiler for NT always ignores the setting of the MOVEABLE, IMPURE, and PRELOAD flags.

3.5.3 LanguageId

The language ID is included in each resource to specify the language that the strings are written with when they need to be translated back to a single byte strings. As well, there may be multiple resources of exactly the same type and name which differ in only the language of the strings within the resources.

The language IDs are documented in Appendix A of the NLS specification, or in winnls.h. The language of a resource or set of resources is specified by the LANGUAGE statement.

3.5.4 Version and Characteristics

Currently, there is space in the resource file format for version and characteristic information of the resource. These values can be set by the resource compiler by using the VERSION or CHARACTERISTICS statements.

3.6 Differentiating 16 and 32-bit resource files.

Because it might be desirable for an ISV's tool that reads and writes resource files to be able to read either the older Windows 16 format files and the new Windows 32 format, Microsoft has devised a method to do this using illegal type and name ordinal numbers.

The method involved is to place an illegal resource in the resource file. The following eight bytes were chosen:

```
0x00 0x00 0x00 0x00 0x20 0x00 0x00 0x00
```

> Assume that it is a 16-bit file. In that case, the Type is illegal since the first 0x00 says string, but a zero-length string is an illegal string. This, then is an illegal 16-bit resource header, indicating that the file is a 32-bit file.

Assume that it is a 32-bit file. Given that, the size of the data is zero, which surely will never be the case.

The Windows 32 resource compiler prefaces each 32-bit resource file with this string of data (followed by an additional data structure describing a zero-length resource with 0 ordinal type and 0 ordinal name), allowing differentiation of 16 and 32-bit resource files. Any tools reading resource files should ignore this resource.

3.7 File Alignment.

Because it is sometimes useful to separate resources into several scripts and then concatenate them after compiling the resource files separatly, it is necessary to specify that resource files are padded to a dword size. If this padding were not included, it could result in the first resource of the second and/or subsequent resource files not aligning upon a dword boundary.

4. Resource Data Format

For any of the pre-defined data types, all structures are DWORD aligned, including the bitmap, icon, and font header structures. As well, the data will always begin on a DWORD boundary.

4.1 Version Resources

Version resources are used to record the version of the application using the resource file. Version resources contain a fixed amount of information. The structure of the version resource is as follows:

```
typedef struct tagVS_FIXEDFILEINFO {
    DWORD       dwSignature;                    // e.g.  0xfeef04bd
    DWORD       dwStrucVersion;                 // e.g.  0x00000042 = "0.42"
    DWORD       dwFileVersionMS;                // e.g.  0x00030075 = "3.75"
    DWORD       dwFileVersionLS;                // e.g.  0x00000031 = "0.31"
    DWORD       dwProductVersionMS;             // e.g.  0x00030010 = "3.10"
    DWORD       dwProductVersionLS;             // e.g.  0x00000031 = "0.31"
    DWORD       dwFileFlagsMask;                // = 0x3F for version "0.42"
    DWORD       dwFileFlags;                    // e.g.  VFF_DEBUG | VFF_PRERELEASE
    DWORD       dwFileOS;                       // e.g.  VOS_DOS_WINDOWS16
    DWORD       dwFileType;                     // e.g.  VFT_DRIVER
    DWORD       dwFileSubtype;                  // e.g.  VFT2_DRV_KEYBOARD
    DWORD       dwFileDateMS;                   // e.g.  0
    DWORD       dwFileDateLS;                   // e.g.  0
    } VS_FIXEDFILEINFO;
```

4.2 Icon Resources

The ICON statement in the .RC script does not create a single resource object, but creates a group of resources. This allows Windows programs a degree of device-independence through the use of different pixel bitmaps on hardware configurations with differing capabilities. Icons, most often designed for differing numbers of pixel planes and pixel counts, are grouped and treated by Windows as a single resource. In the .RES and .EXE files, however, they are stored as a group of resources. These groups are stored in a .RES file with the components first (in this case the different icons [type 3]) and a group header following (Type 14). The group header contains the information necessary to allow Windows to select the proper icon to display.

The components have the following structure:

```
[Resource header (type = 3)]


[DIB Header]
[Color DIBits of icon XOR mask]
[Monochrome DIBits of AND mask]
```

Each component is given an ordinal ID that is unique from all other icon components.

The Device Independent Bitmap (DIB) header's fields represent the masks' information separately with two exceptions. First, the height field represents both the XOR and AND masks. Before converting the two DIBs to Device Dependent Bitmaps (DDB), the height should be divided by two. The masks are always the same size and are one-half the size given in the DIB header. Second, the number of bits per pixel and bit count refer to the XOR mask. The AND mask is always monochrome and should be interpreted as having one plane and one bit per pixel. Before using an icon with Windows refer to the SDK reference materials for more information on DIBs. Since the format of an icon component closely resembles the format of the .ICO file, the documentation in section 9.2 of the Windows SDK Reference is useful. DDBs should not be used for Windows 32 applications.

The group header is described here:

```
[Resource header (type = 14)]


struct IconHeader {
    WORD      wReserved;          // Currently zero
    WORD      wType;              // 1 for icons
    WORD      wCount;             // Number of components
    WORD      padding;            // filler for DWORD alignment
    };
```

The next portion is repeated for each component resource:

```
struct ResourceDirectory {
    BYTE      bWidth;
    BYTE      bHeight;
    BYTE      bColorCount;
    BYTE      bReserved;
    WORD      wPlanes;
    WORD      wBitCount;
    DWORD     lBytesInRes;
    WORD      wNameOrdinal;       // Points to component
    WORD      padding;            // filler for DWORD alignment
    };
```

Notice that the group header consists of a fixed header and data that repeats for each group component. Both of these parts are fixed length allowing for random access of the group component information.

This group header contains all of the data from the .ICO header and from the individual resource descriptors.

4.3 Menu Resources

Menu resources are composed of a menu header followed by a sequential list of menu items. There are two types of menu items: popups and normal menu items. The MENUITEM SEPARATOR is a special case of a normal menu item with an empty name, zero ID, and zero flags. The format for these types is shown here:

[Resource header (type = 4)]

```
struct MenuHeader {
      WORD        wVersion;                        // Currently zero
      WORD        cbHeaderSize;                    // Also zero
      };
```

These next items are repeated for every menu item.

Popup menu items (signalled by fItemFlags & POPUP):

```
struct PopupMenuItem {
      WORD        fItemFlags;
      WCHAR       szItemText[];
      };
```

Normal menu items (signalled by !(fItemFlags & POPUP)):

```
struct NormalMenuItem {
      WORD        fItemFlags;
      WORD        wMenuID;
      WCHAR       szItemText[];
      };
```

The wVersion and cbHeaderSize structure members identify the version of the menu template. They are both zero for Windows 3.0 but may be incremented with future changes to the menu template.

The WORD fItemFlags is a set of flags describing the menu item. If the POPUP bit is set, the item is a POPUP. Otherwise, it is a normal menu component. There are several other flag bits that may be set. Their values are as follows:

```
#define         GRAYED          0x0001  // 'GRAYED' keyword
#define         INACTIVE        0x0002  // 'INACTIVE' keyword
#define         BITMAP          0x0004  // 'BITMAP' keyword
#define         OWNERDRAW       0x0100  // 'OWNERDRAW' keyword
#define         CHECKED         0x0008  // 'CHECKED' keyword
#define         POPUP           0x0010  // Used internally
#define         MENUBARBREAK    0x0020  // 'MENUBARBREAK' keyword
#define         MENUBREAK       0x0040  // 'MENUBREAK' keyword
#define         ENDMENU         0x0080  // Used internally
```

The fItemFlags portion of the last menu item in a given POPUP is flagged by OR'ing it with ENDMENU. It is important to note that since popups can be nested, there may be multiple levels of items with ENDMENU set. When menus are nested, the items are inserted sequentially. A program can traverse this hierarchy by checking for the item with the ENDMENU flag set.

4.4 Dialog Box Resources

A dialog box is contained in a single resource and has a header and a portion repeated for each control in the dialog box. The header is as follows:

```
[Resource header (type = 5)]


struct DialogBoxHeader {
    DWORD       lStyle;
    DWORD       lExtendedStyle;         // new for NT
    WORD        NumberOfItems;
    WORD        x;
    WORD        y;
    WORD        cx;
    WORD        cy;
    [Name or Ordinal] MenuName;
    [Name or Ordinal] ClassName;
    WCHAR       szCaption[];
    WORD        wPointSize;             // Only here if FONT set for dialog
    WCHAR       szFontName[];           // This too
    };
```

The item DWORD lStyle is a standard window style composed of flags found in WINDOWS.H. The default style for a dialog box is:

```
WS_POPUP | WS_BORDER | WS_SYSMENU
```

The lExtendedStyle DWORD is used to specify the extended window style flags. If an extended style is specified on the DIALOG statement, or with the other optional modifier statements, this DWORD is set to that value.

The items marked `Name or Ordinal' are the same format used throughout the resource file (most notably in each resource header) to store a name or an ordinal ID. As before, if the first word is an 0xffff, the next two bytes contain an ordinal ID. Otherwise, the first 1 or more WORDS contain a double-null-terminated string. An empty string is represented by a single WORD zero in the first location.

The WORD wPointSize and WCHAR szFontName entries are present if the FONT statement was included for the dialog box. This can be detected by checking the entry lStyle. if lStyle & DS_SETFONT (DS_SETFONT = 0x40), then these entries will be present.

The data for each control starts on a DWORD boundary (which may require some padding from the previous control), and its format is as follows:

```
struct ControlData {
        DWORD           lStyle;
        DWORD           lExtendedStyle;
        WORD            x;
        WORD            y;
        WORD            cx;
        WORD            cy;
        WORD            wId;
        [Name or Ordinal] ClassId;
        [Name or Ordinal] Text;
        WORD            nExtraStuff;
        };
```

As before, the item DWORD lStyle is a standard window style composed of the flags found in WINDOWS.H. The type of control is determined by the class. The class is either given by a zero-terminated string, or in the case of many common Windows classes, is given a one word code to save space and speed processing - in this case, the ordinal number will be a WORD in length, but only the lower byte will be used. Because UNICODE allows 0x8000 as a legal character, the ordinal classes are prefaced with a of 0xFFFF, similar to the ordinal Type and Name fields. The one byte classes are listed here:

```
#define                 BUTTON                          0x80
#define                 EDIT                            0x81
#define                 STATIC                          0x82
#define                 LISTBOX                         0x83
#define                 SCROLLBAR                       0x84
#define                 COMBOBOX                        0x85
```

The lExtendedStyle DWORD is used to specify the extended style flags to be used for this control. The extended style flags are placed at the end of the CONTROL (or other control statements) statement following the coordinates

The extra information at the end of the control data structure is currently not used, but is intended for extra information that may be needed for menu items in the future. Usually it is zero length.

The various statements used in a dialog script are all mapped to these classes along with certain modifying styles. The values for these styles can be found in WINDOWS.H. All dialog controls have the default styles of WS_CHILD and WS_VISIBLE. A list of the default styles used to make the script statements follows:

Statement	Default Class	Default Styles
CONTROL	None	WS_CHILD|WS_VISIBLE
LTEXT	STATIC	ES_LEFT
RTEXT	STATIC	ES_RIGHT
CTEXT	STATIC	ES_CENTER
LISTBOX	LISTBOX	WS_BORDER | LBS_NOTIFY
CHECKBOX	BUTTON	BS_CHECKBOX | WS_TABSTOP
PUSHBUTTON	BUTTON	BS_PUSHBUTTON | WS_TABSTOP
GROUPBOX	BUTTON	BS_GROUPBOX
DEFPUSHBUTTON	BUTTON	BS_DEFPUSHBUTTON | WS_TABSTOP
RADIOBUTTON	BUTTON	BS_RADIOBUTTON
AUTOCHECKBOX	BUTTON	BS_AUTOCHECKBOX
AUTO3STATE	BUTTON	BS_AUTO3STATE
AUTORADIOBUTTON	BUTTON	BS_AUTORADIOBUTTON
PUSHBOX	BUTTON	BS_PUSHBOX
STATE3	BUTTON	BS_3STATE
EDITTEXT	EDIT	ES_LEFT|WS_BORDER|WS_TABSTOP
COMBOBOX	COMBOBOX	None
ICON	STATIC	SS_ICON
SCROLLBAR	SCROLLBAR	None

The control text is stored in the `Name or Ordinal' format described in detail above.

4.5 Cursor Resources

Cursor resources are very much like icon resources. They are formed in groups with the components preceding the header. This header also employs a fixed-length component index that allows random access of the individual components. The structure of the cursor header is as follows:

[Resource header (type = 12)]

```
struct CursorHeader {
      WORD        wReserved;                       // Currently zero
      WORD        wType;                           // 2 for cursors
      WORD        cwCount;                         // Number of components
      WORD        padding;                         // filler for DWORD alignment
      };
```

The next portion is repeated for each component resource, and starts on a DWORD boundary.

```
struct ResourceDirectory {
      WORD        wWidth;
      WORD        wHeight;
      WORD        wPlanes;
      WORD        wBitCount;
      DWORD       lBytesInRes;
      WORD        wNameOrdinal;                    // Points to component
      WORD        padding;                         // filler for DWORD alignment
      };
```

Each cursor component is also similar to each icon component. There is, however, one significant difference between the two: cursors have the concept of a `hotspot' where icons do not. Here is the component structure:

[Resource header (Type = 1)]

```
struct CursorComponent {
        short           xHotspot;
        short           yHotspot;
        }
             [Monochrome XOR mask]
             [Monochrome AND mask]
```

These masks are bitmaps copied from the .CUR file. The main difference from icons in this regard is that cursors do not have color DIBs used for XOR masks like cursors. Although the bitmaps are monochrome and do not have DIB headers or color tables, the bits are still in DIB format with respect to alignment and direction. See the SDK Reference for more information on DIB formats.

4.6 Bitmap Resources

Windows 32 can read two types of device-independent bitmaps. The normal type of DIB is the Windows 3.0 DIB format. The other type of DIB is that used for OS/2 versions 1.1 and 1.2. The bitmap resource consists of a single device-independent bitmap and accordingly, this DIB can be of either format. The two DIBs are distinguished by their header structures. They both have the size of their respective structures as the first DWORD in the header. Both these structures are documented in the Windows SDK Reference Version 3.0 volume 2, section 7. The header structure for the normal DIB is BITMAPINFOHEADER while the OS/2 DIB header is called BITMAPCOREHEADER. The correct size (as a DWORD) must be in the first entry of the structure.

```
[Normal resource header (type = 2)]


[BITMAPINFOHEADER or BITMAPCOREHEADER]
[Color table if not 24 bits per pixel]
[Packed-pixel bitmap]
```

Note that the color table is optional. All but 24 bit color bitmaps have a color table attached next. This table's length can be computed by 2#BitsPerPixel * 3 bytes for OS/2 bitmaps or 2#BitsPerPixel * 4 bytes for Windows bitmaps. The bitmap image data is placed immediately following the color table.

Note that the BItmap file has an unaligned header structure (BITMAPFILEHEADER structure). This header is not, however, stored in the resource file, as it serves only to identify the type of file (DIB or DDB)

4.7 Font and Font Directory Resources

Font resources are different from the other types of resources in that they are not added to the resources of a specific application program. Font resources are added to .EXE files that are renamed to be .FON files. These files are libraries as opposed to applications.

Font resources use a resource group structure. Individual fonts are the components of a font group. Each component is defined by a FONT statement in the .RC file. The group header follows all components and contains all information necessary to access a specific font. The format of a font component resource is as follows:

```
[Normal resource header (type = 8)]
```

```
[Complete contents of the .FNT file follow as the resource body -- See the Windows SDK Reference for the format of the .FNT file]
```

The group header for the fonts is normally last in the .RES file. Note that unlike cursor and icon groups, the font group need not be contiguous in the .RES file. Font declarations may be placed in the .RC file mixed with other resource declarations. The group header is added automatically by RC at the end of the .RES file. Programs generating .RES files must add the FONTDIR entry manually. The group header has the following structure:

```
[Normal resource header (type = 7)]
```

```
WORD NumberOfFonts;// Total number in .RES file
```

The remaining data is repeated for every font in the .RES file.

```
WORD fontOrdinal;
struct FontDirEntry {
    WORD        dfVersion;
    DWORD       dfSize;
    char        dfCopyright[60];
    WORD        dfType;
    WORD        dfPoints;
    WORD        dfVertRes;
    WORD        dfHorizRes;
    WORD        dfAscent;
    WORD        dfInternalLeading;
    WORD        dfExternalLeading;
    BYTE        dfItalic;
    BYTE        dfUnderline;
    BYTE        dfStrikeOut;
    WORD        dfWeight;
    BYTE        dfCharSet;
    WORD        dfPixWidth;
    WORD        dfPixHeight;
    BYTE        dfPitchAndFamily;
    WORD        dfAvgWidth;
    WORD        dfMaxWidth;
    BYTE        dfFirstChar;
    BYTE        dfLastChar;
    BYTE        dfDefaultChar;
    BYTE        dfBreakChar;
    WORD        dfWidthBytes;
    DWORD       dfDevice;
    DWORD       dfFace;
    DWORD       dfReserved;
    char        szDeviceName[];
    char        szFaceName[];
    };
```

4.8 String Table Resources

These tables are constructed in blocks of 16 strings. The organization of these blocks of 16 is determined by the IDs given to the various strings. The lowest four bits of the ID determine a string's position in the block. The upper twelve bits determine which block the string is in. Each block of 16 strings is stored as one resource entry. Each string or error table resource block is stored as follows:

[Normal resource header (type = 6 for strings)]

[Block of 16 strings. The strings are Pascal style with a WORD length preceding the string. 16 strings are always written, even if not all slots are full. Any slots in the block with no string have a zero WORD for the length.]

It is important to note that the various blocks need not be written out in numerical order in the resource file. Each block is assigned an ordinal ID. This ID is the high 12 bits of the string IDs in the block plus one (ordinal IDs can't be zero). The blocks are written to the .RES file in the order the blocks are encountered in the .RC file, while the CVTRES utility will cause them to become ordered in the COFF object, and hence the image file.

4.9 Accelerator Table Resources

An accelerator table is stored as a single resource. Multiple accelerator tables are also allowed. The format of an accelerator table is very simple. No header for the table is used. Each entry in the table has a single five-byte entry. The last entry in the table has its flag word's high bit set (fFlags |= 0x8000). Since all entries are fixed length, random access can be done because the number of elements in the table can be computed by dividing the length of the resource by eight. Here is the structure of the table entries:

[Normal resource header (type = 9)]

The following structure is repeated for all accelerator table entries.

```
struct AccelTableEntry {
    WORD        fFlags;
    WORD        wAscii;
    WORD        wId;
    WORD        padding;
    };
```

4.10 User Defined Resources and RCDATA

RC allows the programmer to include resources not defined in Windows. The user may choose a name not defined as a standard type and use it to include data that is to be used as a resource. This data may be taken from an external file or may be placed between BEGIN and END statements. As an option, the programmer can define the type as RCDATA with the same results.

As might be imagined, the format of this resource is very simple because the resource compiler knows nothing about the logical structure of the data. Here is the organization:

[Normal resource header (type = 10 for RCDATA, named types represent user-defined types)]

[The data from the BEGIN ... END or from the external file is included without translation into the .RES file]

4.11 Name Table and Error Table Resources

Name tables and error resources are no longer supported in the Windows binary resource file format.

4.12 Version Resources.

Version resources specify information that can be used by setup programs to discover which of several versions of a program or dynamic link library to install into the system. There is also a set of api's to query the version resources. There are three major types of information stored in version resources: the main information in a VS_FIXEDFILEINFO structure, Language information data in a variable file information structure (VarFileInfo), and user defined string information in StringFileInfo structures. For Windows 32, the strings within the version information resource is stored in Unicode, providing localization of the resoruces. Each block of information is dword aligned.

The structure of a version resource is depicted by the VS_VERSION_INFO structure.

```
VS_VERSION_INFO {

    WORD wLength;                   /* Length of the version resource */
    WORD wValueLength;              /* Length of the value field for this block */
    WORD wType;                     /* type of information:  1==string, 0==binary */
    WCHAR szKey[];                  /* Unicode string KEY field */
    [WORD Padding1;]                /* possible word of padding */
    VS_FIXEDFILEINFO Value;         /* Fixed File Info Structure */
    BYTE Children[];                /* position of VarFileInfo or StringFileInfo data */
};
```

The Fixed File Info structure contains basic information about the version, including version numbers for the product and file, and type of the file.

```
typedef struct tagVS_FIXEDFILEINFO {
    DWORD dwSignature;              /* signature - always 0xfeef04bd */
    DWORD dwStrucVersion;           /* structure version - currently 0 */
    DWORD dwFileVersionMS;          /* Most Significant file version dword */
    DWORD dwFileVersionLS;          /* Least Significant file version dword */
    DWORD dwProductVersionMS;       /* Most Significant product version */
    DWORD dwProductVersionLS;       /* Least Significant product version */
    DWORD dwFileFlagMask;                   /* file flag mask */
    DWORD dwFileFlags;              /*  debug/retail/prerelease/... */
    DWORD dwFileOS;                 /* OS type.  Will always be Windows32 value */
```

```
   DWORD dwFileType;                /* Type of file (dll/exe/drv/... )*/
   DWORD dwFileSubtype;             /* file subtype */
   DWORD dwFileDateMS;              /* Most Significant part of date */
   DWORD dwFileDateLS;              /* Least Significant part of date */
} VS_FIXEDFILEINFO;
```

The user defined string information is contained within the StringFileInfo structure, which is a set of two strings: the key and the information itself.

```
StringFileInfo {

   WCHAR       szKey[];             /* Unicode "StringFileInfo" */
   [WORD       padding;]            /* possible padding */
   StringTable Children[];          /*
};

StringTable {

   WCHAR     szKey[];               /* Unicode string denoting the language - 8 bytes */
   String Children[];               /* array of children String structures */
}

String {
   WCHAR   szKey[];                 /* arbitrary Unicode encoded KEY string */
                                    /* note that there is a list of pre-defined keys */
   [WORD   padding;]                         /* possible padding */
   WCHAR Value[];                   /* Unicode-encoded value for KEY */
} String;
```

The variable file info (VarFileInfo) block contains a list of languages supported by this version of the application/dll.

```
VarFileInfo {

   WCHAR     szKey[];               /* Unicode "VarFileInfo" */
   [WORD     padding;];             /* possible padding */
   Var       Children[];            /* children array */
};

Var {
   WCHAR     szKey[];               /* Unicode "Translation" (or other user key) */
   [WORD     padding;]              /* possible padding */
   WORD      Value[];               /* one or more values, normally language id's */
};
```

4.13 Messagetable Resources.

A message table is a resource that contains formatted text that is used to display an error message or messagebox. It has taken the place of the error table resource (which was never used). The data consists of a MESSAGE_RESOURCE_DATA structure, which contains one or more MESSAGE_RESOURCE_BLOCKS, which in turn may consist of one or more MESSAGE_RESOURCE_ENTRY structures. The structure is similar to that of the STRINGTABLE resource.

```
typedef struct _MESSAGE_RESOURCE_DATA {

   ULONG NumberOfBlocks;
   MESSAGE_RESOURCE_BLOCK Blocks[];
} MESSAGE_RESOURCE_DATA, *PMESSAGE_RESOURCE_DATA;

typedef struct _MESSAGE_RESOURCE_BLOCK {
   ULONG LowId;
   ULONG HighId;
   ULONG OffsetToEntries;
} MESSAGE_RESOURCE_BLOCK, *PMESSAGE_RESOURCE_BLOCK;

typedef struct _MESSAGE_RESOURCE_ENTRY {
   USHORT Length;
   USHORT Flags;
   UCHAR Text[];
} MESSAGE_RESOURCE_ENTRY, *PMESSAGE_RESOURCE_ENTRY;
```