Resource Hacker

Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

Introduction:

Resource Hacker has been designed to:

1. **View** resources in Win32 executable and related files (*.exe, *.dll, *.cpl, *.ocx) in both their compiled and decompiled formats.

2. **Extract** (save) resources to file in (*.res) format, as a binary, or as decompiled resource scripts or images.
Icons, bitmaps, cursors, menus, dialogs, string tables, message tables, accelerators, Borland forms and version info resources can be fully decompiled into their respective formats, whether as image files or *.rc text files.

3. **Modify** (replace) resources in executables.
Image resources (icons, cursors and bitmaps) can be replaced with an image from a corresponding image file (*.ico, *.cur, *.bmp), a *.res file or even another *.exe file.
Dialogs, menus, stringtables, accelerators and messagetable resource scripts (and also Borland forms) can be edited and recompiled using the internal resource script editor.
Resources can also be replaced with resources from a *.res file as long as the replacement resource is of the same type and has the same name.

4. **Add** new resources to executables.
Enable a program to support multiple languages, or add a custom icon or bitmap (company logo etc) to a program's dialog.

5. **Delete** resources. Most compilers add resources into applications which are never used by the application. Removing these unused resources can reduce an application's size.

Resource Hacker
Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

Installation:

"Resource Hacker" requires no installation apart from extracting the executable and the accompanying documentation files from the zip file into the desired folder. No entries are made in the Window's Registry. In order to store information between sessions, an *ini file* will be created in the application's folder.

Uninstall:

To remove the program, simply delete these files.

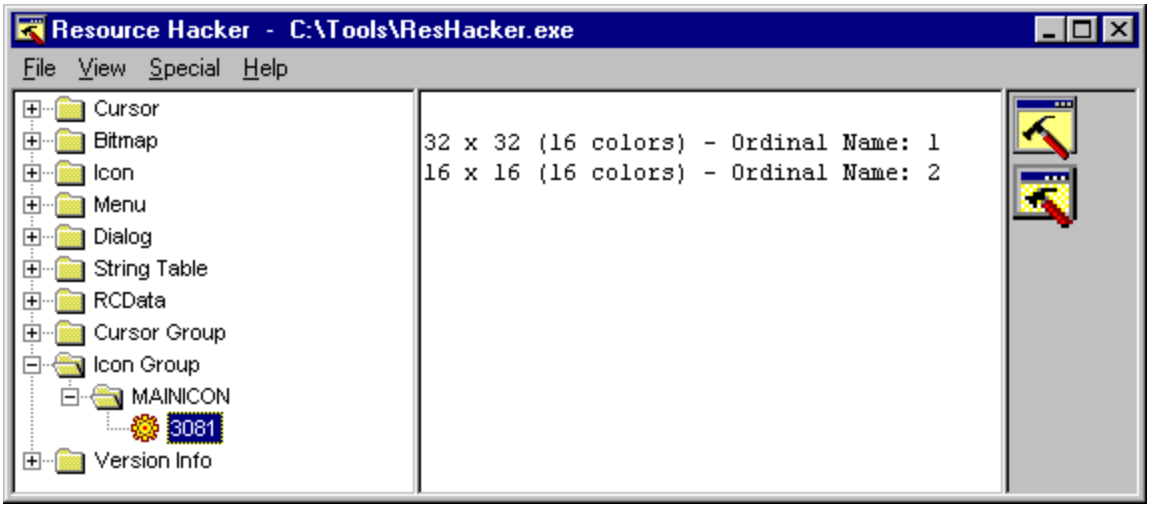
Resource Hacker

Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

Overview:

Any Windows 95/98/NT executable files (including 32bit exe's, dll's, ocx's and cpl's) can be opened by selecting `File|Open` from Resource Hacker's menu. A full list of the file's resources will be displayed in a tree structure. The resource tree can be fully expanded or collapsed by selecting `View|Expand Tree` or `View|Collapse Tree` respectively from the menu.



A specific resource item is defined by its resource type, name and languageID.

Resources are grouped into "**resource types**". There are a number of pre-defined resource types (icons, cursors, bitmaps, dialogs, menus, rcdata etc) but the programmer may also have defined other resource types.

Resource items are stored within their respective resource types and have a "**resource name**" which is unique within that type. This 'name' can usually be either an integer value or an alphanumeric string, however, some resource types (eg stringtables) allow only integer values for names.

Each named resource can have more than one language specific item to enable programs to handle multiple languages. Under each resource name in the resource tree there will appear at least one "**resource language**" item. The languageID is a word integer value made up of a primary language byte and a sublanguage byte which is

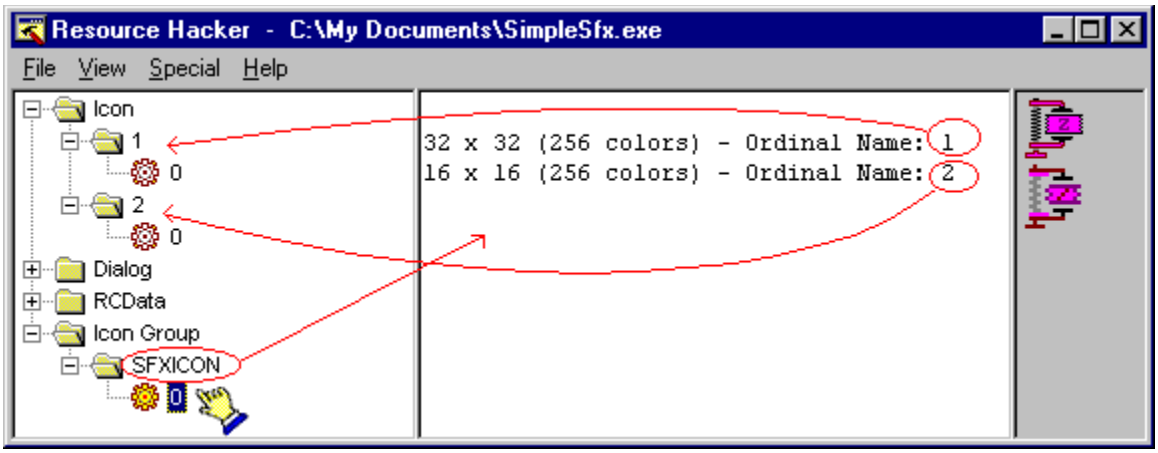
defined by Windows. (If the resource item is “language neutral” then this value will be zero.)

Cursors and Icons:

Cursors and icons require special mention here as their resource information is split over 2 resource types: “Cursor” & “Cursor Group”; and “Icon” & “Icon Group” respectively.

Each icon (or cursor) can have several images - eg 16x16 (16 colors), 32x32 (16 colors), 16x16 (256 colors). The image that windows actually renders depends on variables such as - small icon, large icon, display color resolution etc.

As an example, the windows command LOADICON() will first find the icon info with the designated IconName within the “Icon Group”. This icon info contains the number of images and the “Icon” type NameOrdinals (word integer ‘names’) of all the images pertaining to the designated icon. Windows then renders the “best” image (if more than one exists) for the current windows configuration.



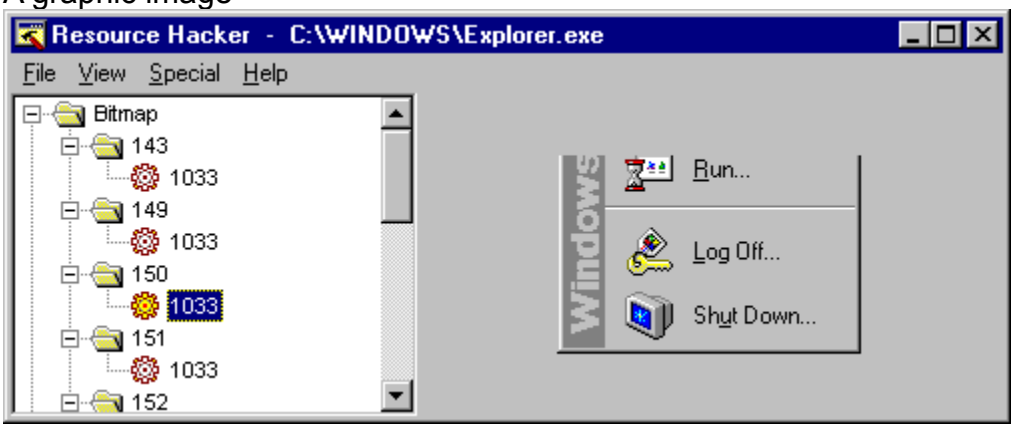
When *replacing* icons (or cursors) in a file, the “whole” icon is replaced: the icon info (in “Icon Group”) *and* respective images (in “Icon”).

When *extracting (saving)* icons (or cursors) from a resource: if an “Icon” type item is highlighted in the resource tree then the image will be saved as a single image icon. Howeve, if an “Icon Group” item is highlighted then all the images pertaining to the selected icon group will be saved to the specified icon file.

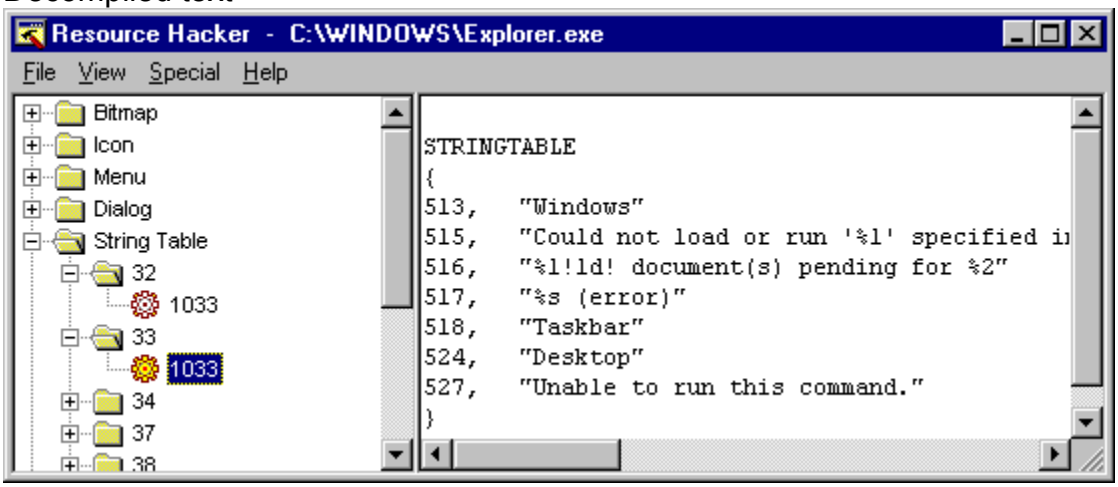
Viewing resources:

Simply select the resource from the resource tree (once a file has been opened). The resource will displayed either as:

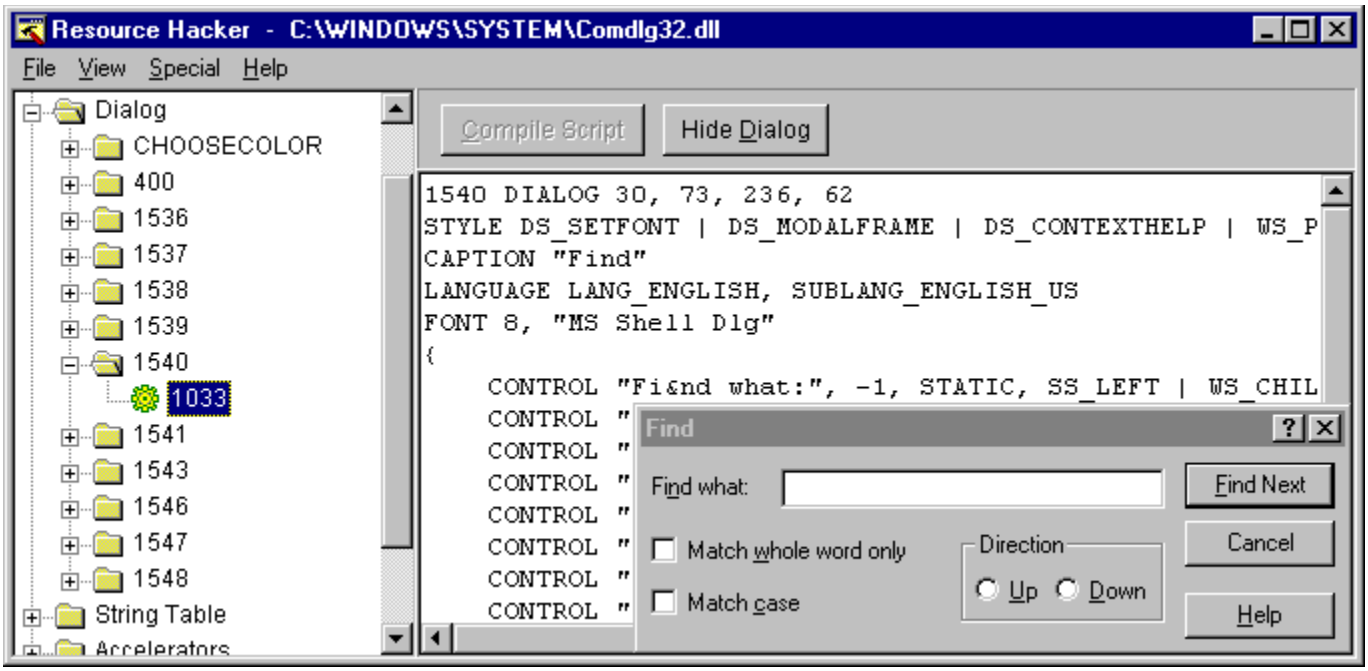
A graphic image -



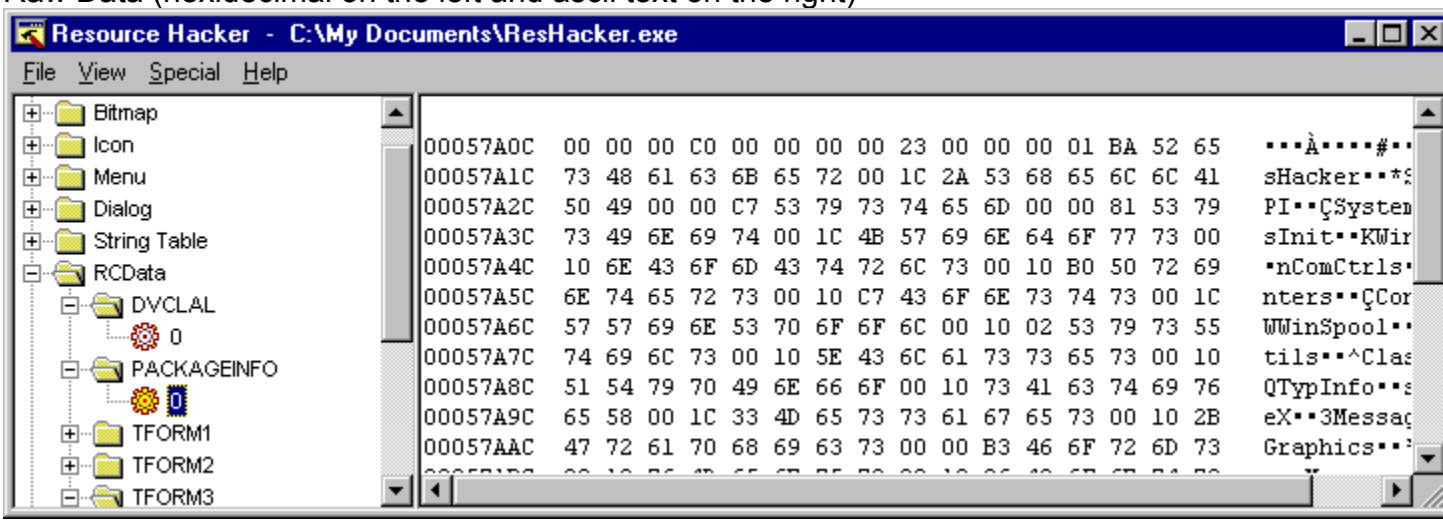
Decompiled text -



A combination of a compiled image and decompiled text -



Raw Data (hexidecimal on the left and ascii text on the right) -



Note: Programs compiled using the Borland VCL (eg most Delphi programs) do not commonly have dialog, menu or accelerator resources, but store this information in "RCData".

Extracting (saving) resources:

Icons, bitmaps, cursors, menus, dialogs, string tables, message tables, accelerators, Borland forms and version info resources can all be saved to file in their uncompiled formats, whether as image files or as resource text files (*.rc). Having first selected the

specific resource using the treeview control, select `Special | Save [Resource Name] ...` from the menu. To save a resource to a compiled resource file (*.res): select `Special | Save Resource as a *.res file ...` from the menu. To save a single resource as a raw binary file: select `Special | Save Resource as a Binary file ...` from the menu.

To save to file all resources of a specific resource type, first select the specific resource type using the treeview control. Then select `Special | Save [Resource Type] Resources ...` from the menu.

To save to file *all* resources, select `Special | Save all Resources...` from the menu. Resources which cannot be decompiled into resource script (eg: images) are converted back into their original file formats and stored in the same folder as the resource script. (eg Cursor.cur, Cursor_2.cur, Bitmap.bmp, Icon.ico). Resources with unrecognised data formats are stored as *.bin files.

Modifying (replacing) resources:

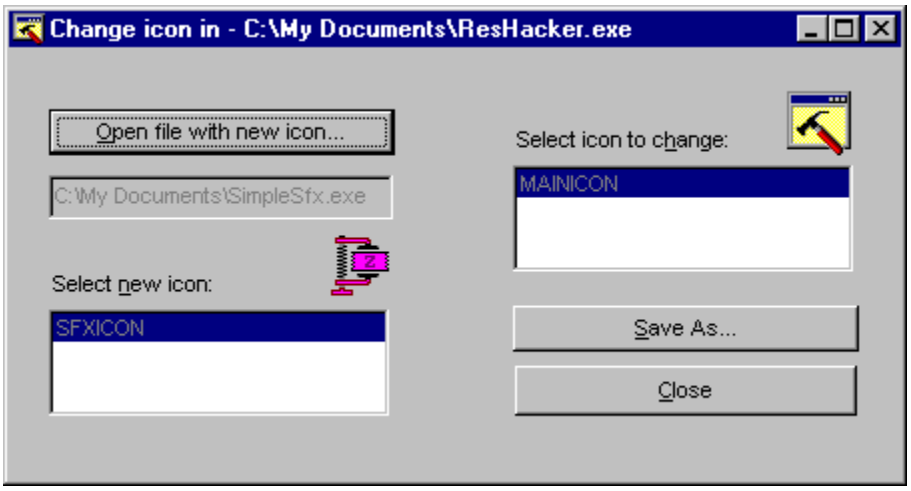
Warnings:

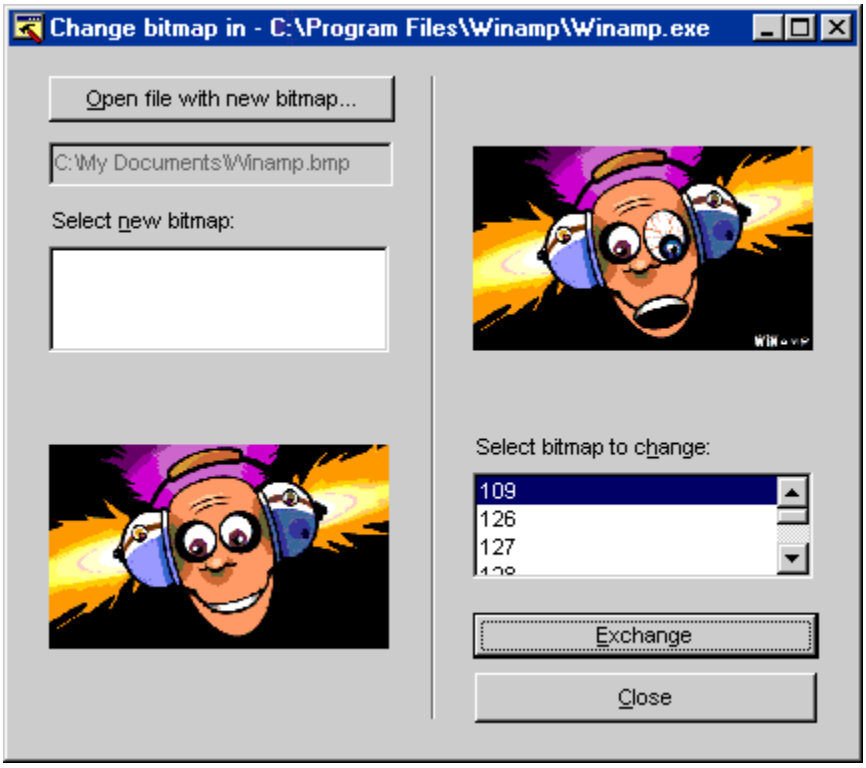
If the user intends to modify resources, make sure the original file is *backed up* first. Then, thoroughly test the 'new' file before discarding the backup.

With dialog and menu resources, deleting controls or changing control IDs is likely to cause the modified program to crash. However, changing a control's caption is usually safe, as is modifying their position, size and visibility. Adding a new control is unlikey to cause problems.

Replacing Images:

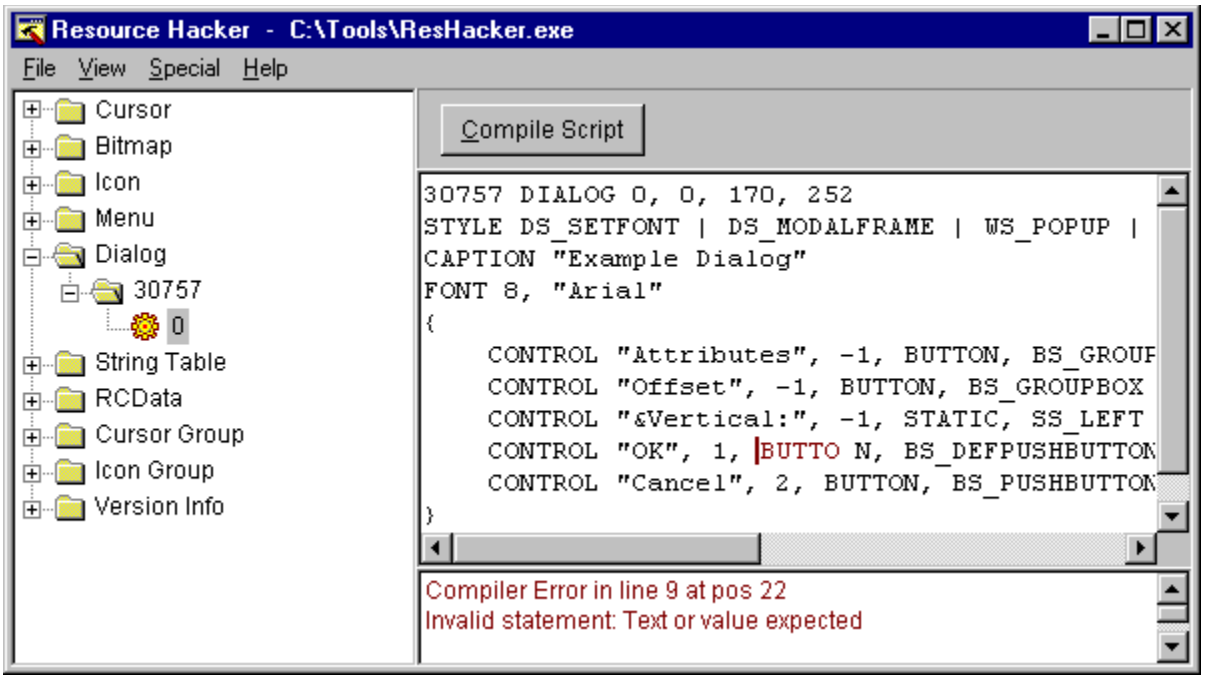
If the resource item to be replaced is an icon, cursor, or a bitmap the source can be an *.ico, *.cur or *.bmp file respectively or selected from a *.res or another *.exe file. Select `Special | Replace Icon` (Cursor or Bitmap) from the menu.





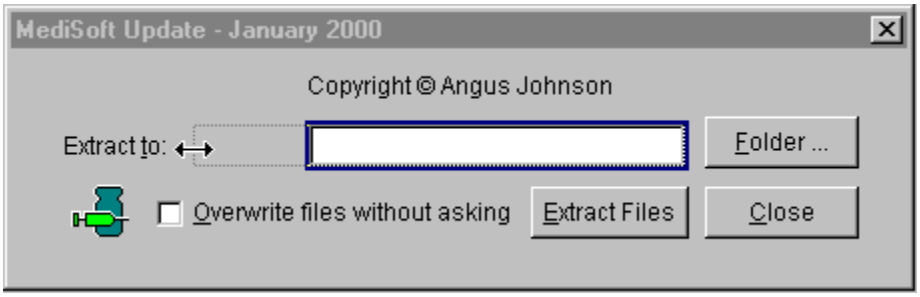
Using the internal editor to modify text based resources:

Dialog, menu, stringtable, messagetable, accelerators and Borland form resources can all be easily edited and recompiled using the internal resource editor. The internal compiler supports - `\t` , `\n` , `\\` , `\”` , and `\000` .. `\377` - in resource strings to represent *tab*, *newline, backslash, doublequote* and *octal* bytes respectively. The - #define - statement is also supported. Simply edit the displayed resource script, and click the [Compile] button. The modified compiled resource will then be displayed. Any errors encountered during compilation will be reported with an error message.

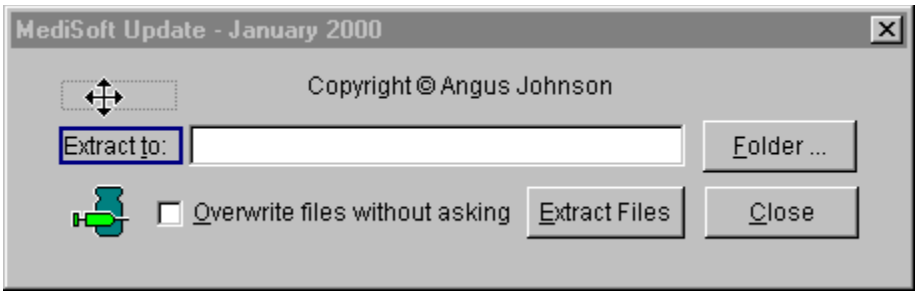


Dialog controls can also be visually resized and or moved, with any changes being reflected in the resource script automatically. Conversion between screen pixels and dialog units is done automatically.

First select a control by clicking it in the displayed dialog. The borders of the control can then be dragged to resize the control.



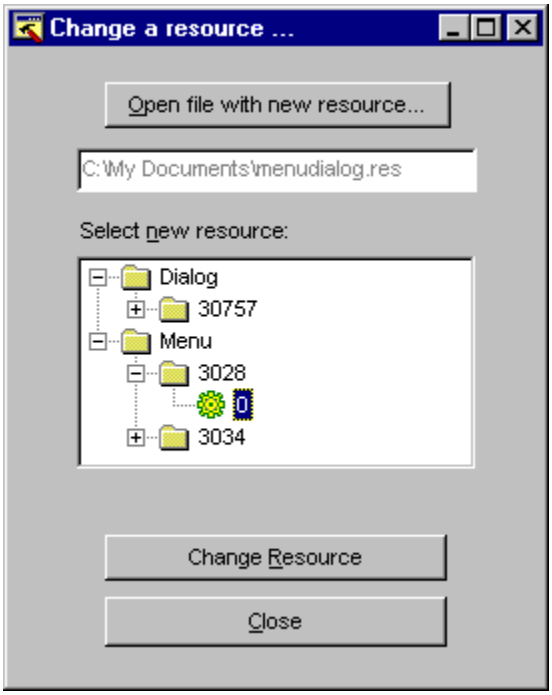
To move the control without resizing, click on the control *after* it has been selected and drag it to its new location.



Once this is done the script needs to be recompiled before saving. To select a new control either click on it with the mouse, or *tab* to it with the tab key. If a control cannot be clicked, then it has probably been "covered" by another control. Controls which are declared in the resource script below other controls are "drawn" on top of preceeding controls if their positions overlap. To move or resize a "hidden" control, either move or shrink the overlapping control or change the order the controls have been declared in the resource script.

Replacing resources other than images:

Resources can also be replaced with resources located in external *.res files. Replacement resources must not only be of the same resource type but must also have the same resource name. Select `Special | Replace other Resource ...` from the menu.



Updating all resources with resources in a *.res file:

Resources can also be replaced with **all the matching** resources located in an external *.res file. A resource in the *res file will only replace a resource in the target (exe,dll) file if it has the same resource type, name and language id. Select `Special | Update all Resources ...` from the menu.

Once all modifications have been made, the modified file image can be saved to file by selecting `File | Save As` from the menu.

Adding resources:

Resources can be added to an executable as long as no resource of the same type, name and language id already exists. Select `Special | Add a New Resource ...` from the menu. The new resource can only be added from a *.res file.

Adding resources can enable a program to **support multiple languages**.
(Note: Windows95 and Windows98 do not use multiple language resources, this is a WindowsNT feature. I presume Windows98 just picks the first resource if there is more than one language resource available.)

As an example:
* Task: Add a FRENCH translation of Dialog 30757 to Samples.dll.
* Solution:
1. Open Samples.dll and select Dialog 30757.
Note in the script the language is currently LANG_NEUTRAL, SUBLANG_NEUTRAL.
2. Change this to LANG_FRENCH, SUBLANG_FRENCH, and translate the dialog caption and each control caption into French. Click [Compile].
3. Save the resource to a *.res file (eg: FrenchDlg30757.res).
4. Close Samples.dll *without* saving and then reopen it.
Dialog 30757 should still be LANG_NEUTRAL, SUBLANG_NEUTRAL.
5. Select `Special | Add New Resource` from the menu and open the file FrenchDlg30757.res which has just been created.
6. Select Dialog 30757 which now has the languageId 1036 (French) and click [Add Resource].
7. Finally, save Samples.dll. Voila!

Tip: Changing Resource Hacker's editor font can help when viewing scripts in different languages. (Select `View | Editor Font ...` from the menu.)

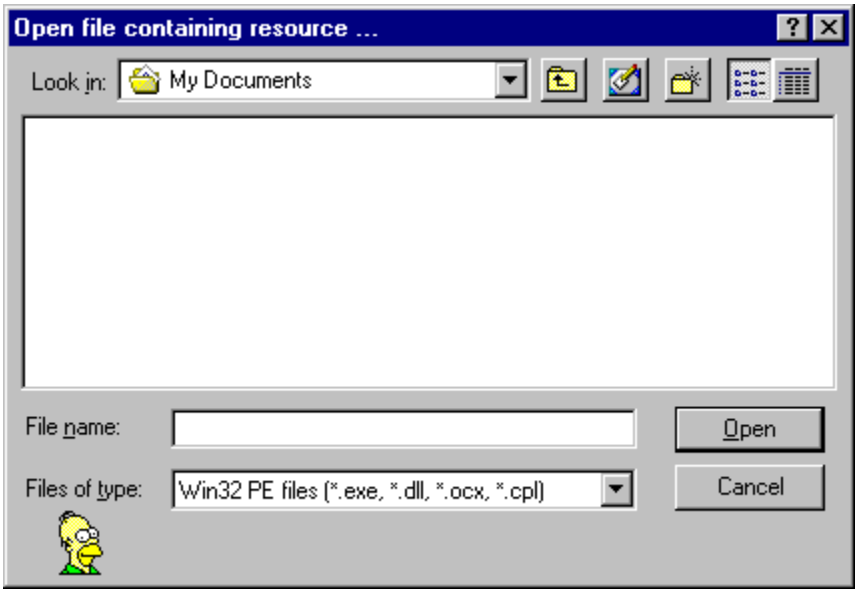
Adding icons or bitmaps (company logos etc) enables them to be displayed in the program's dialogs.

As an example:
* Task: Add an icon to Windows' FileOpen dialog.
* Solution:

1. Open “c:\windows\system\comdlg32.dll”.
(The FileOpen dialog is located in this dll and is ‘named’ 1547.)
2. It’s the ‘Icon Group’ type we’re interested in here, so make sure the ‘Icon Group’ name for the icon to be added (from a *.res file) has not already been used. In ComDlg32.dll, ‘names’ in the Icon Group range from 528 to 539, so any other integer value or alphanumeric string can be used. (Don’t worry about the resource names used in the ‘Icon’ type as Resource Hacker will make sure that the icon images associated with the new icon will also be given unique names.)
(*Tip: if you don’t have a resource compiler to create the *.res file containing the new icon, replace Resource Hacker’s MAINICON with the desired icon from an *.ico file and then save it as a *.res file.*)
3. Add the icon to the dll by Selecting `Special` | `Add New Resource` from the menu.
4. Assuming the new icon is named ‘MAINICON’, add the following control at the end of the control list in the Dialog named 1547:

```
CONTROL “MAINICON”,-1,STATIC, SS_ICON|WS_CHILD|WS_VISIBLE,13,142,21,20
```

5. Compile the dialog script and save the file as comdlgXX.dll.
6. The comdlg32.dll cannot be replaced while Windows is running, it can only be replaced in DOS mode. Shut down Windows and restart in MS_DOS mode.
7. Use the DOS commands to rename the the old comdlg32.dll something like comdlg98.dll and then rename the newly created file comdlg32.dll.
8. Restart windows. That’s it. (Of course, adding a bitmap instead of an icon is less confusing because there is no ‘Bitmap Group’ for bitmaps.)



Resource Hacker
Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

History:

10 September 2000 (Version 2.5):

* Resources can now also be deleted (except VersionInfo resources).
* Bug Fix: Modified applications occasionally displayed the generic executable icon, not the application's icon.
* Fixes to a couple of other minor bugs.

18 August 2000 (Version 2.4.0.4):

* Bug Fix: Internal compiler would not compile some Chinese text.
* Improved translation support. Scripts are now compiled using the codepage appropriate to the internal editor's selected font rather than the operating system's default codepage.
(Thanks to Frank Cheng for feedback while fixing both these DBCS issues.)
* Bug fix: Occasionally StringTable resources would not be decompiled.
* Bug fix: Accelerators would not compile if they included the ASCII keyword.
* The command line will now accept a filename as a parameter.
* The *Samples.dll* file is no longer included in the download.
* New homepage: http://rpi.net.au/~ajohnson/resourcehacker.

3 July 2000 (Version 2.4.0.3):

* Bug Fix: Bug introduced with changes in the previous update which caused an error in "Update all Resources" preventing any updates.
* Bug Fix: Occasional bug when extracting resources to a RES file.
* A couple of very minor improvements to the compiler have also been made.

26 May 2000 (Version 2.4.0.2):

* Multibyte character set support (Chinese, Japanese, Korean) for the internal editor has been added (with thanks to Bob Ishida for feedback during debugging).
* Bug fix: Cursors with multiple images were not being correctly imported when replacing cursors.
* Numerous other improvements and cosmetic changes.
* A number of documentation errors in this help file have also been fixed.

20 Apr 2000 (Version 2.3.0.6):

* The JPG and MIDI data formats are now detected and displayed or played.
* Bug Fix: WAVE, AVI and GIF formats were not being detected in the RCDATA section.
* Bug Fix: The folder where resource data was last saved was not being stored between sessions.

16 Apr 2000 (Version 2.3.0.5):

* The AVI and WAVE data formats are now detected and displayed or played.
* The GIF data format is now detected and displayed (with thanks to Anders Melander for TGifImage).
* A number of sample resources have been removed from the Resource Hacker executable and placed in "Samples.dll". *(Note: ''Samples.dll'' is no longer included in the download.)*

13 Feb 2000 (Version 2.3.0.3):

* The editor's font can now be changed. This also enables changes to the editor's font script (character code) which is useful when translating resources.
* Menu editor [Show/Hide] button added.
* Coloured treeview cursor removed.
* Dialog controls now have coordinates displayed when selected too (see version 2.3.0.2).

11 Feb 2000 (Version 2.3.0.2):

* Bug Fix: Major modifications to dll's still occasionally failed. Now finally fixed.
* The treeview window width can now be adjusted.
* Resource Hacker's window size and position is now stored between sessions (in an *ini file) as are the folders for the last opened and saved files.
* While moving or resizing dialog controls - the control coordinates (in dialog units) are now displayed in the panel located above the dialog script.

9 Feb 2000 (Version 2.3.0.1):

* LANGUAGE statements in resource scripts are no longer "read-only".
* Resources can now be updated with all matching resources in an external resource file (*.res) in a single operation.
* Resources can now be added.
* Bug Fix: Replacing cursors & icons from *.res files was broken.

1 Feb 2000 (Version 2.2.0.1):

* Bug Fix: Modifying dll's occasionally stopped them working (relative virtual addresses of sections following the resource section were not being adjusted).
***** MESSAGETABLE and ACCELERATOR resources can now also be edited and recompiled using the internal editor.
* Dialogs will be displayed even when they contain unregistered controls (a gray rectangle will appear in the position of each unregistered control).

* The dialog editor now compiles controls defined by using either of the following styles:
    CONTROL text, control-ID, control-class, control-style, x, y, width, height
    CLASS_MAIN_STYLE text, control-ID, x, y, width, height, control-style
* Resources (menus, dialogs, stringtables, accelerators & messagetables) can now be saved in a single operation to a single *.rc file.
* Modified files now preserve the original file date and time.
* Numerous other minor improvements.

17 Jan 2000 (Version 2.1.1.4):

* Bug Fix: Memory leak fixed.
* Bug Fix: Numeric captions in dialog resources were not compiled correctly.
* Resources can now be saved to file as a binary.
* Improved handling of special characters (tab, newline, backslash & doublequote) in resource scripts.
* Dialog resource forms and controls can now be visually moved and resized.

2 Jan 2000 (Version 2.0.1.2):

* DIALOG, DIALOGEX, MENU, MENUEX, STRINGTABLE and BORLAND FORM resources can now be edited and recompiled using the internal editor.
* Bug Fix - large resources which were displayed as hexidecimal took forever to load. The display algorithm for these resources is now *much* faster.
* Bug Fix - Tab and Newline characters are now converted to \t and \n respectively in dialog script control captions.
***** Bitmap Exchange Dialog added - enables viewing of bitmaps while selecting.
* Individual icons are no longer scaled but are displayed at their actual size.
* Files can now be opened by dragging them into Resource Hacker.

12 Dec 1999 (version 1.0.0.5):

***** Bug Fix - Exchanging cursors from *.cur files was broken.
***** Bug Fix - Menu resource scripts did not always decompile correctly.
***** Icon & Cursor Exchange Dialogs now display selected images.
***** Icons & cursor resources are now hidden in the 'Exchange Other Resources' dialog.
***** MENUEX and DIALOGEX resource scripts are now properly supported.
***** Resource scripts now decompile control style attributes too.
* Borland Delphi form files now decompiled.
* Accelerators now decompiled.
* Help file added.

03 Dec 1999 (version 0.5.0.1):

***** Initial Release.

Resource Hacker
Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

Known Limitations:

Issue 1:
Resource Hacker has been compiled using Delphi™ ver 3.02.
When decompiling and recompiling Borland's Delphi forms in applications compiled with Delphi ver 5.0, there will be errors in the recompiled application if frames have been used to create the form. This error is due to the *inline* DFM keyword not being recognised. While decompiling, the *inline* keyword will be replaced by *object* and, if manually corrected before recompiling, *inline* will be rejected by the internal compiler.

To solve this limitation, a fair amount of work will be required in order to successfully compile Resource Hacker using Delphi ver 5.0.

Issue 2:
A number of applications have been "packed" with an EXE packer after they have been compiled to reduce the size of a program. This has a side-effect of making it much more difficult to view and modify resources. When a "packed" executable is viewed with *Resource Hacker*, only resource types and names will be visible but not the actual resources.

This is not viewed as a bug. The application developer may well have viewed this as beneficial feature so no "fix" is planned.

Resource Hacker

Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

Licence Agreement:

This program has been released as freeware under the following conditions:

1. It is not to be distributed via the internet or via any other media without the prior approval of the author. In particular, it is not to be made available from internet sites which promote the illegal modification of software.

2. It is not used in such a way as to modify the copyright notice of this or any other software, to in any way disguise the registered user or owner of any software, or to in any way illegally modify or breach the copyright of any software.

3. No guarantee of performance is given. Any damage to software resulting from using ‘‘Resource Hacker’’ will be the responsibility of the user.

Please send any bug reports to the author at the above email address. All feedback is welcome.

Resource Hacker

Version 2.5

ajohnson@rpi.net.au
http://rpi.net.au/~ajohnson/resourcehacker

“Pleeeeease show me the source code!”

Dozens of people have emailed me requesting information on how to modify executables. Although this program is freeware, I’m not releasing the source code. So *please* don’t ask for it! However, for anyone who has an interest in this topic the following info should help get you started:

1. Borland Delphi’s demo - Resource Explorer.
It has a few bugs (its menu decompiling algorithm inparticular) but is still a very useful starting point.

2. Borland’s command line utility TDump.exe (distributed with Delphi & Cbuilder++).

3. MSDN - *http://msdn.microsoft.com/default.asp*

4. A good hex editor.

5. The layout of Win32 executables (exe’s, dll’s, ocx’s, cpl’s etc) have a specific format. The following info was downloaded from *http://www.wotsit.org*. (*http://www.wotsit.org* is an excellent resource for many file formats.)

```
This text is copyright 1999 by B. Luevelsmeyer.
It is freeware, and you may use it for any purpose but on your own risk.
bernd.luevelsmeyer@iplan.heitec.net


The PE file format
==================


Preface
-------

The PE ("portable executable") file format is the format of executable
binaries (DLLs and programs) for MS windows NT, windows 95 and win32s; in
windows NT, the drivers are in this format, too.
It can also be used for object files and libraries.

The format is designed by Microsoft and standardized by the TIS (tool
interface standard) Committee (Microsoft, Intel, Borland, Watcom, IBM and
others) in 1993, apparently based on a good knowledge of COFF, the
```

"common object file format" used for object files and executables on several UNIXes and on VMS.

The win32 SDK includes a header file <winnt.h> containing #defines and typedefs for the PE-format. I will mention the struct-member-names and #defines as we go.

You may also find the DLL "imagehelp.dll" to be helpful. It is part of windows NT, but documentation is scarce. Some of its functions are described in the "Developer Network".

General Layout

At the start of a PE file we find an MS-DOS executable ("stub"); this makes any PE file a valid MS-DOS executable.

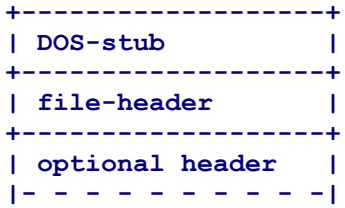
After the DOS-stub there is a 32-bit-signature with the magic number 0x00004550 (IMAGE_NT_SIGNATURE).

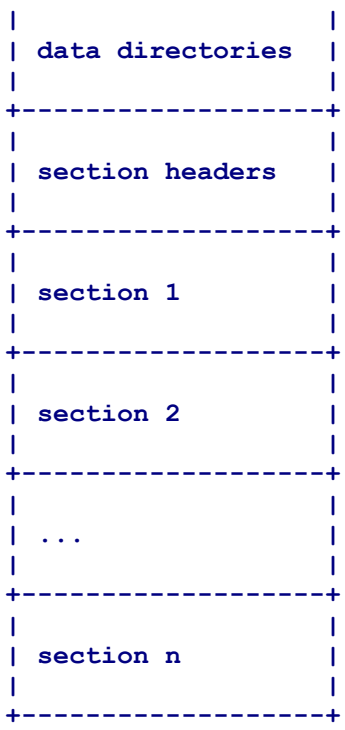
Then there is a file header (in the COFF-format) that tells on which machine the binary is supposed to run, how many sections are in it, the time it was linked, whether it is an executable or a DLL and so on. (The difference between executable and DLL in this context is: a DLL can not be started but only be used by another binary, and a binary cannot link to an executable).

After that, we have an optional header (it is always there but still called "optional" - COFF uses an "optional header" for libraries but not for objects, that's why it is called "optional"). This tells us more about how the binary should be loaded: The starting address, the amount of stack to reserve, the size of the data segment etc..

An interesting part of the optional header is the trailing array of 'data directories'; these directories contain pointers to data in the 'sections'. If, for example, the binary has an export directory, you will find a pointer to that directory in the array member IMAGE_DIRECTORY_ENTRY_EXPORT, and it will point into one of the sections.

Following the headers we find the 'sections', introduced by the 'section headers'. Essentially, the sections' contents is what you really need to execute a program, and all the header and directory stuff is just there to help you find it. Each section has some flags about alignment, what kind of data it contains ("initialized data" and so on), whether it can be shared etc., and the data itself. Most, but not all, sections contain one or more directories referenced through the entries of the optional header's "data directory" array, like the directory of exported functions or the directory of base relocations. Directoryless types of contents are, for example, "executable code" or "initialized data".





DOS-stub and Signature

The concept of a DOS-stub is well-known from the 16-bit-windows- executables (which were in the "NE" format). The stub is used for OS/2-executables, self-extracting archives and other applications, too. For PE-files, it is a MS-DOS 2.0 compatible executable that almost always consists of about 100 bytes that output an error message such as "this program needs windows NT". You recognize a DOS-stub by validating the DOS-header, being a struct IMAGE_DOS_HEADER. The first 2 bytes should be the sequence "MZ" (there is a #define IMAGE_DOS_SIGNATURE for this WORD). You distinguish a PE binary from other stubbed binaries by the trailing signature, which you find at the offset given by the header member 'e_lfanew' (which is 32 bits long beginning at byte offset 60). For OS/2 and windows binaries, the signature is a 16-bit-word; for PE files, it is a 32-bit-longword aligned at a 8-byte-boundary and having the value IMAGE_NT_SIGNATURE #defined to be 0x00004550.

File Header

To get to the IMAGE_FILE_HEADER, validate the "MZ" of the DOS-header (1st 2 bytes), then find the 'e_lfanew' member of the DOS-stub's header and skip that many bytes from the beginning of the file. Verify the signature you will find there. The file header, a struct IMAGE_FILE_HEADER, begins immediately after it; the members are described top to bottom.

The first member is the 'Machine', a 16-bit-value indicating the system the binary is intended to run on. Known legal values are

IMAGE_FILE_MACHINE_I386 (0x14c)
    for Intel 80386 processor or better

0x014d
    for Intel 80486 processor or better

0x014e
    for Intel Pentium processor or better

0x0160
    for R3000 (MIPS) processor, big endian

IMAGE_FILE_MACHINE_R3000 (0x162)
    for R3000 (MIPS) processor, little endian

IMAGE_FILE_MACHINE_R4000 (0x166)
    for R4000 (MIPS) processor, little endian

IMAGE_FILE_MACHINE_R10000 (0x168)
    for R10000 (MIPS) processor, little endian

IMAGE_FILE_MACHINE_ALPHA (0x184)
    for DEC Alpha AXP processor

IMAGE_FILE_MACHINE_POWERPC (0x1F0)
    for IBM Power PC, little endian

Then we have the 'NumberOfSections', a 16-bit-value. It is the number of sections that follow the headers. We will discuss the sections later.

Next is a timestamp 'TimeDateStamp' (32 bit), giving the time the file was created. You can distinguish several versions of the same file by this value, even if the "official" version number was not altered. (The format of the timestamp is not documented except that it should be somewhat unique among versions of the same file, but apparently it is 'seconds since January 1 1970 00:00:00' in UTC - the format used by most C compilers for the time_t.) This timestamp is used for the binding of import directories, which will be discussed later. Warning: some linkers tend to set this timestamp to absurd values which are not the time of linking in time_t format as described.

The members 'PointerToSymbolTable' and 'NumberOfSymbols' (both 32 bit) are used for debugging information. I don't know how to decipher them, and I've found the pointer to be always 0.

'SizeOfOptionalHeader' (16 bit) is simply sizeof(IMAGE_OPTIONAL_HEADER). You can use it to validate the correctness of the PE file's structure.

'Characteristics' is 16 bits and consists of a collection of flags, most of them being valid only for object files and libraries:

> Bit 0 (IMAGE_FILE_RELOCS_STRIPPED) is set if there is no relocation information in the file. This refers to relocation information per section in the sections themselves; it is not used for executables, which have relocation information in the 'base relocation' directory described below.

Bit 1 (IMAGE_FILE_EXECUTABLE_IMAGE) is set if the file is executable, i.e. it is not an object file or a library. This flag may also be set if the linker attempted to create an executable but failed for some reason, and keeps the image in order to do e.g. incremental linking the next time. Bit 2 (IMAGE_FILE_LINE_NUMS_STRIPPED) is set if the line number information is stripped; this is not used for executable files.

Bit 3 (IMAGE_FILE_LOCAL_SYMS_STRIPPED) is set if there is no information about local symbols in the file (this is not used for executable files).

Bit 4 (IMAGE_FILE_AGGRESIVE_WS_TRIM) is set if the operating system is supposed to trim the working set of the running process (the amount of RAM the process uses) aggressivly by paging it out. This should be set if it is a demon-like application that waits most of the time and only wakes up once a day, or the like.

Bits 7 (IMAGE_FILE_BYTES_REVERSED_LO) and 15 (IMAGE_FILE_BYTES_REVERSED_HI) are set if the endianess of the file is not what the machine would expect, so it must swap bytes before reading. This is unreliable for executable files (the OS expects executables to be correctly byte-ordered).

Bit 8 (IMAGE_FILE_32BIT_MACHINE) is set if the machine is expected to be a 32 bit machine. This is always set for current implementations; NT5 may work differently.

Bit 9 (IMAGE_FILE_DEBUG_STRIPPED) is set if there is no debugging information in the file. This is unused for executable files. According to other information ([6]), this bit is called "fixed" and is set if the image can only run if it is loaded at the preferred load address (i.e. it is not relocatable).

Bit 10 (IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP) is set if the application may not run from a removable medium such as a floppy or a CD-ROM. In this case, the operating system is advised to copy the file to the swapfile and execute it from there.

Bit 11 (IMAGE_FILE_NET_RUN_FROM_SWAP) is set if the application may not run from the network. In this case, the operating system is advised to copy the file to the swapfile and execute it from there.

Bit 12 (IMAGE_FILE_SYSTEM) is set if the file is a system file such as a driver. This is unused for executable files; it is also not used in all the NT drivers I inspected.

Bit 13 (IMAGE_FILE_DLL) is set if the file is a DLL.

Bit 14 (IMAGE_FILE_UP_SYSTEM_ONLY) is set if the file is not designed to run on multiprocessor systems (that is, it will crash there because it relies in some way on exactly one processor).

Relative Virtual Addresses

The PE format makes heavy use of so-called RVAs. An RVA, aka "relative

virtual address", is used to describe a memory address if you don't know the base address. It is the value you need to add to the base address to get the linear address. The base address is the address the PE image is loaded to, and may vary from one invocation to the next.

Example: suppose an executable file is loaded to address 0x400000 and execution starts at RVA 0x1560. The effective execution start will then be at the address 0x401560. If the executable were loaded to 0x100000, the execution start would be 0x101560.

Things become complicated because the parts of the PE-file (the sections) are not necessarily aligned the same way the loaded image is. For example, the sections of the file are often aligned to 512-byte-borders, but the loaded image is perhaps aligned to 4096-byte-borders. See 'SectionAlignment' and 'FileAlignment' below.

So to find a piece of information in a PE-file for a specific RVA, you must calculate the offsets as if the file were loaded, but skip according to the file-offsets. As an example, suppose you knew the execution starts at RVA 0x1560, and want to diassemble the code starting there. To find the address in the file, you will have to find out that sections in RAM are aligned to 4096 bytes and the ".code"-section starts at RVA 0x1000 in RAM and is 16384 bytes long; then you know that RVA 0x1560 is at offset 0x560 in that section. Find out that the sections are aligned to 512-byte-borders in the file and that ".code" begins at offset 0x800 in the file, and you know that the code execution start is at byte 0x800+0x560=0xd60 in the file.

Then you disassemble and find an access to a variable at the linear address 0x1051d0. The linear address will be relocated upon loading the binary and is given on the assumption that the preferred load address is used. You find out that the preferred load address is 0x100000, so we are dealing with RVA 0x51d0. This is in the data section which starts at RVA 0x5000 and is 2048 bytes long. It begins at file offset 0x4800. Hence. the veriable can be found at file offset 0x4800+0x51d0-0x5000=0x49d0.

Optional Header

Immediately following the file header is the IMAGE_OPTIONAL_HEADER (which, in spite of the name, is always there). It contains information about how to treat the PE-file exactly. We'll also have the members from top to bottom.

The first 16-bit-word is 'Magic' and has, as far as I looked into PE-files, always the value 0x010b.

The next 2 bytes are the version of the linker ('MajorLinkerVersion' and 'MinorLinkerVersion') that produced the file. These values, again, are unreliable and do not always reflect the linker version properly. (Several linkers simply don't set this field.) And, coming to think about it, what good is the version if you have got no idea *which* linker was used?

The next 3 longwords (32 bit each) are intended to be the size of the executable code ('SizeOfCode'), the size of the initialized data ('SizeOfInitializedData', the so-called "data segment"), and the size of the uninitialized data ('SizeOfUninitializedData', the so-called "bss segment"). These values are, again, unreliable (e.g. the data segment may actually be

split into several segments by the compiler or linker), and you get better sizes by inspecting the 'sections' that follow the optional header.

Next is a 32-bit-value that is a RVA. This RVA is the offset to the codes's entry point ('AddressOfEntryPoint'). Execution starts here; it is e.g. the address of a DLL's LibMain() or a program's startup code (which will in turn call main()) or a driver's DriverEntry(). If you dare to load the image "by hand", you call this address to start the process after you have done all the fixups and the relocations.

The next 2 32-bit-values are the offsets to the executable code ('BaseOfCode') and the initialized data ('BaseOfData'), both of them RVAs again, and both of them being of little interest because you get more reliable information by inspecting the 'sections' that follow the headers. There is no offset to the uninitialized data because, being uninitialized, there is little point in providing this data in the image.

The next entry is a 32-bit-value giving the preferred (linear) load address ('ImageBase') of the entire binary, including all headers. This is the address (always a multiple of 64 KB) the file has been relocated to by the linker; if the binary can in fact be loaded to that address, the loader doesn't need to relocate the file again, which is a win in loading time. The preferred load address can not be used if another image has already been loaded to that address (an "address clash", which happens quite often if you load several DLLs that are all relocated to the linker's default), or the memory in question has been used for other purposes (stack, malloc(), uninitialized data, whatever). In these cases, the image must be loaded to some other address and it needs to be relocated (see 'relocation directory' below). This has further consequences if the image is a DLL, because then the "bound imports" are no longer valid, and fixups have to be made to the binary that uses the DLL - see 'import directory' below.

The next 2 32-bit-values are the alignments of the PE-file's sections in RAM ('SectionAlignment', when the image has been loaded) and in the file ('FileAlignment'). Usually both values are 32, or FileAlignment is 512 and SectionAlignment is 4096. Sections will be discussed later.

The next 2 16-bit-words are the expected operating system version ('MajorOperatingSystemVersion' and 'MinorOperatingSystemVersion' [they _do_ like self-documenting names at MS]). This version information is intended to be the operating system's (e.g. NT or Win95) version, as opposed to the subsystem's version (e.g. Win32); it is often not supplied, or wrong supplied. The loader doesn't use it, apparently.

The next 2 16-bit-words are the binary's version, ('MajorImageVersion' and 'MinorImageVersion'). Many linkers don't set this information correctly and many programmers don't bother to supply it, so it is better to rely on the version-resource if one exists.

The next 2 16-bit-words are the expected subsystem version ('MajorSubsystemVersion' and 'MinorSubsystemVersion'). This should be the Win32 version or the POSIX version, because 16-bit-programs or OS/2-programs won't be in PE-format, obviously. This subsystem version should be supplied correctly, because it *is* checked and used: If the application is a Win32-GUI-application and runs on NT4, and the subsystem version is *not* 4.0, the dialogs won't be 3D-style and certain other features will also work "old-style" because the application expects to run on NT 3.51, which had the

program manager instead of explorer and so on, and NT 4.0 will mimic that behaviour as faithfully as possible.

Then we have a 'Win32VersionValue' of 32 bits. I don't know what it is good for. It has been 0 in all the PE files that I inspected.

Next is a 32-bits-value giving the amount of memory the image will need, in bytes ('SizeOfImage'). It is the sum of all headers' and sections' lengths if aligned to 'SectionAlignment'. It is a hint to the loader how many pages it will need in order to load the image.

The next thing is a 32-bit-value giving the total length of all headers including the data directories and the section headers ('SizeOfHeaders'). It is at the same time the offset from the beginning of the file to the first section's raw data.

Then we have got a 32-bit-checksum ('CheckSum'). This checksum is, for current versions of NT, only checked if the image is a NT-driver (the driver will fail to load if the checksum isn't correct). For other binary types, the checksum need not be supplied and may be 0. The algorithm to compute the checksum is property of Microsoft, and they won't tell you. However, several tools of the Win32 SDK will compute and/or patch a valid checksum, and the function CheckSumMappedFile() in the imagehelp.dll will do so too. The checksum is supposed to prevent loading of damaged binaries that would crash anyway - and a crashing driver would result in a BSOD, so it is better not to load it at all.

Then there is a 16-bit-word 'Subsystem' that tells in which of the NT-subsystems the image runs:

IMAGE_SUBSYSTEM_NATIVE (1)
: The binary doesn't need a subsystem. This is used for drivers.

IMAGE_SUBSYSTEM_WINDOWS_GUI (2)
: The image is a Win32 graphical binary. (It can still open a console with AllocConsole() but won't get one automatically at startup.)

IMAGE_SUBSYSTEM_WINDOWS_CUI (3)
: The binary is a Win32 console binary. (It will get a console per default at startup, or inherit the parent's console.)

IMAGE_SUBSYSTEM_OS2_CUI (5)
: The binary is a OS/2 console binary. (OS/2 binaries will be in OS/2 format, so this value will seldom be used in a PE file.)

IMAGE_SUBSYSTEM_POSIX_CUI (7)
: The binary uses the POSIX console subsystem.

Windows 95 binaries will always use the Win32 subsystem, so the only legal values for these binaries are 2 and 3; I don't know if "native" binaries on windows 95 are possible.

The next thing is a 16-bit-value that tells, if the image is a DLL, when to call the DLL's entry point ('DllCharacteristics'). This seems not to be used; apparently, the DLL is always notified about everything.

If bit 0 is set, the DLL is notified about process attachment (i.e. DLL load).

If bit 1 is set, the DLL is notified about thread detachments (i.e. thread terminations).

If bit 2 is set, the DLL is notified about thread attachments (i.e. thread creations).

If bit 3 is set, the DLL is notified about process detachment (i.e. DLL unload).

The next 4 32-bit-values are the size of reserved stack ('SizeOfStackReserve'), the size of initially committed stack ('SizeOfStackCommit'), the size of the reserved heap ('SizeOfHeapReserve') and the size of the committed heap ('SizeOfHeapCommit'). The 'reserved' amounts are address space (not real RAM) that is reserved for the specific purpose; at program startup, the 'committed' amount is actually allocated in RAM. The 'committed' value is also the amount by which the committed stack or heap grows if necessary. (Other sources claim that the stack will grow in pages, regardless of the 'SizeOfStackCommit' value. I didn't check this.) So, as an example, if a program has a reserved heap of 1 MB and a committed heap of 64 KB, the heap will start out at 64 KB and is guaranteed to be enlargeable up to 1 MB. The heap will grow in 64-KB-chunks. The 'heap' in this context is the primary (default) heap. A process can create more heaps if so it wishes. The stack is the first thread's stack (the one that starts main()). The process can create more threads which will have their own stacks. DLLs don't have a stack or heap of their own, so the values are ignored for their images. I don't know if drivers have a heap or a stack of their own, but I don't think so.

After these stack- and heap-descriptions, we find 32 bits of 'LoaderFlags', which I didn't find a useful description of. I only found a vague note about setting bits that automatically invoke a breakpoint or a debugger after loading the image; however, this doesn't seem to work.

Then we find 32 bits of 'NumberOfRvaAndSizes', which is the number of valid entries in the directories that follow immediately. I've found this value to be unreliable; you might wish use the constant IMAGE_NUMBEROF_DIRECTORY_ENTRIES instead, or the lesser of both.

After the 'NumberOfRvaAndSizes' there is an array of IMAGE_NUMBEROF_DIRECTORY_ENTRIES (16) IMAGE_DATA_DIRECTORYs. Each of these directories describes the location (32 bits RVA called 'VirtualAddress') and size (also 32 bit, called 'Size') of a particular piece of information, which is located in one of the sections that follow the directory entries. For example, the security directory is found at the RVA and has the size that are given at index 4. The directories that I know the structure of will be discussed later. Defined directory indexes are:

IMAGE_DIRECTORY_ENTRY_EXPORT (0)
The directory of exported symbols; mostly used for DLLs. Described below.

IMAGE_DIRECTORY_ENTRY_IMPORT (1)
The directory of imported symbols; see below.

IMAGE_DIRECTORY_ENTRY_RESOURCE (2)
Directory of resources. Described below.

IMAGE_DIRECTORY_ENTRY_EXCEPTION (3)
Exception directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_SECURITY (4)
Security directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_BASERELOC (5)
Base relocation table - see below.

IMAGE_DIRECTORY_ENTRY_DEBUG (6)
Debug directory - contents is compiler dependent. Moreover, many compilers stuff the debug information into the code section and don't create a separate section for it.

IMAGE_DIRECTORY_ENTRY_COPYRIGHT (7)
Description string - some arbitrary copyright note or the like.

IMAGE_DIRECTORY_ENTRY_GLOBALPTR (8)
Machine Value (MIPS GP) - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_TLS (9)
Thread local storage directory - structure unknown; contains variables that are declared "__declspec(thread)", i.e. per-thread global variables.

IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG (10)
Load configuration directory - structure and purpose unknown.

IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11)
Bound import directory - see description of import directory.

IMAGE_DIRECTORY_ENTRY_IAT (12)
Import Address Table - see description of import directory.

As an example, if we find at index 7 the 2 longwords 0x12000 and 33, and the load address is 0x10000, we know that the copyright data is at address 0x10000+0x12000 (in whatever section there may be), and the copyright note is 33 bytes long. If a directory of a particular type is not used in a binary, the Size and VirtualAddress are both 0.

Section directories

The sections consist of two major parts: first, a section description (of type IMAGE_SECTION_HEADER) and then the raw section data. So after the data directories we find an array of 'NumberOfSections' section headers, ordered by the sections' RVAs.

A section header contains:

An array of IMAGE_SIZEOF_SHORT_NAME (8) bytes that make up the name (ASCII) of the section. If all of the 8 bytes are used there is no 0- terminator for the string! The name is typically something like ".data" or ".text" or

".bss". There need not be a leading '.', the names may also be "CODE" or "IAT" or the like. Please note that the names are not at all related to the contents of the section. A section named ".code" may or may not contain the executable code; it may just as well contain the import address table; it may also contain the code *and* the address table *and* the initialized data. To find information in the sections, you will have to look it up via the data directories of the optional header. Do not rely on the names, and do not assume that the section's raw data starts at the beginning of a section.

The next member of the IMAGE_SECTION_HEADER is a 32-bit-union of 'PhysicalAddress' and 'VirtualSize'. In an object file, this is the address the contents is relocated to; in an executable, it is the size of the contents. In fact, the field seems to be unused; There are linkers that enter the size, and there are linkers that enter the address, and I've also found a linker that enters a 0, and all the executables run like the gentle wind.

The next member is 'VirtualAddress', a 32-bit-value holding the RVA to the section's data when it is loaded in RAM.

Then we have got 32 bits of 'SizeOfRawData', which is the size of the secion's data rounded up to the next multiple of 'FileAlignment'.

Next is 'PointerToRawData' (32 bits), which is incredibly useful because it is the offset from the file's beginning to the section's data. If it is 0, the section's data are not contained in the file and will be arbitrary at load time.

Then we have got 'PointerToRelocations' (32 bits) and 'PointerToLinenumbers' (also 32 bits), 'NumberOfRelocations' (16 bits) and 'NumberOfLinenumbers' (also 16 bits). All of these are information that's only used for object files. Executables have a special base relocation directory, and the line number information, if present at all, is usually contained in a special purpose debugging segment or elsewhere.

The last member of a section header is the 32 bits 'Characteristics', which is a bunch of flags describing how the section's memory should be treated:

> If bit 5 (IMAGE_SCN_CNT_CODE) is set, the section contains executable code.
>
> If bit 6 (IMAGE_SCN_CNT_INITIALIZED_DATA) is set, the section contains data that gets a defined value before execution starts. In other words: the section's data in the file is meaningful.
>
> If bit 7 (IMAGE_SCN_CNT_UNINITIALIZED_DATA) is set, this section contains uninitialized data and will be initialized to all-0-bytes before execution starts. This is normally the BSS.
>
> If bit 9 (IMAGE_SCN_LNK_INFO) is set, the section doesn't contain image data but comments, description or other documentation. This information is part of an object file and may be information for the linker, such as which libraries are needed.
>
> If bit 11 (IMAGE_SCN_LNK_REMOVE) is set, the data is part of an object file's section that is supposed to be left out when the executable file is linked. Often combined with bit 9.

If bit 12 (IMAGE_SCN_LNK_COMDAT) is set, the section contains "common block data", which are packaged functions of some sort.

If bit 15 (IMAGE_SCN_MEM_FARDATA) is set, we have far data - whatever that means. This bit's meaning is unsure.

If bit 17 (IMAGE_SCN_MEM_PURGEABLE) is set, the section's data is purgeable - but I don't think that this is the same as "discardable", which has a bit of its own, see below. The same bit is apparently used to indicate 16-bit-information as there is also a define IMAGE_SCN_MEM_16BIT for it. This bit's meaning is unsure.

If bit 18 (IMAGE_SCN_MEM_LOCKED) is set, the section should not be moved in memory? Perhaps it indicates there is no relocation information? This bit's meaning is unsure.

If bit 19 (IMAGE_SCN_MEM_PRELOAD) is set, the section should be paged in before execution starts? This bit's meaning is unsure.

Bits 20 to 23 specify an alignment that I have no information about. There are #defines IMAGE_SCN_ALIGN_16BYTES and the like. The only value I've ever seen used is 0, for the default 16-byte- alignment. I suspect that this is the alignment of objects in a library file or the like.

If bit 24 (IMAGE_SCN_LNK_NRELOC_OVFL) is set, the section contains some extended relocations that I don't know about.

If bit 25 (IMAGE_SCN_MEM_DISCARDABLE) is set, the section's data is not needed after the process has started. This is the case, for example, with the relocation information. I've seen it also for startup routines of drivers and services that are only executed once, and for import directories.

If bit 26 (IMAGE_SCN_MEM_NOT_CACHED) is set, the section's data should not be cached. Don't ask my why not. Does this mean to switch off the 2nd-level-cache?

If bit 27 (IMAGE_SCN_MEM_NOT_PAGED) is set, the section's data should not be paged out. This is interesting for drivers.

If bit 28 (IMAGE_SCN_MEM_SHARED) is set, the section's data is shared among all running instances of the image. If it is e.g. the initialized data of a DLL, all running instances of the DLL will at any time have the same variable contents. Note that only the first instance's section is initialized. Sections containing code are always shared copy-on-write (i.e. the sharing doesn't work if relocations are necessary).

If bit 29 (IMAGE_SCN_MEM_EXECUTE) is set, the process gets 'execute'-access to the section's memory.

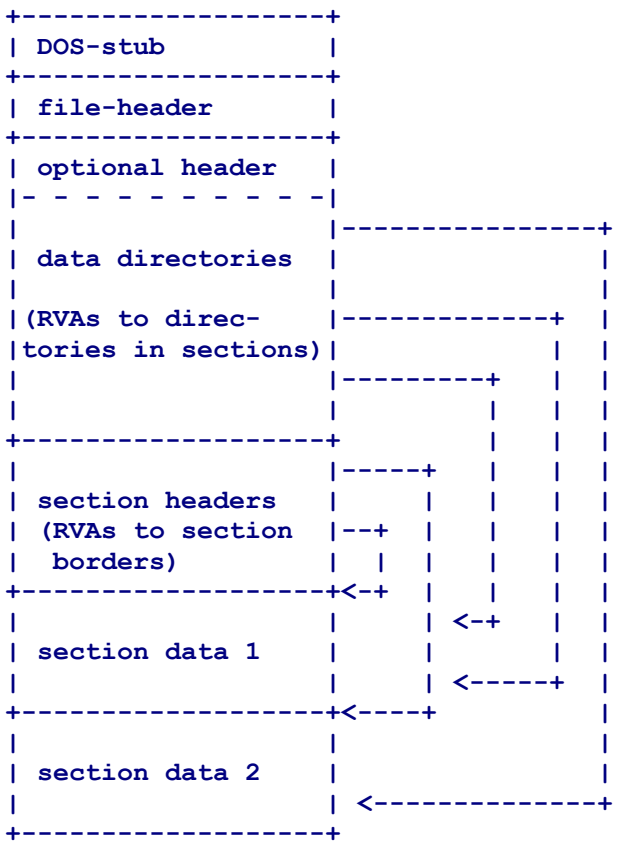
If bit 30 (IMAGE_SCN_MEM_READ) is set, the process gets 'read'-access to the section's memory.

If bit 31 (IMAGE_SCN_MEM_WRITE) is set, the process gets 'write'-access to the section's memory.

After the section headers we find the sections themselves. They are, in the file, aligned to 'FileAlignment' bytes (that is, after the optional header and after each section's data there will be padding bytes) and ordered by their RVAs. When loaded (in RAM), the sections are aligned to 'SectionAlignment' bytes.

As an example, if the optional header ends at file offset 981 and 'FileAlignment' is 512, the first section will start at byte 1024. Note that you can find the sections via the 'PointerToRawData' or the 'VirtualAddress', so there is hardly any need to actually fuss around with the alignments.

I will try to make an image of it all:



There is one section header for each section, and each data directory will point to one of the sections (several data directories may point to the same section, and there may be sections without data directory pointing to them).

Sections' raw data

general

All sections are aligned to 'SectionAlignment' when loaded in RAM, and 'FileAlignment' in the file. The sections are described by entries in the section headers: You find the sections in the file via 'PointerToRawData' and in memory via 'VirtualAddress'; the length is in 'SizeOfRawData'.

There are several kinds of sections, depending on what's contained in them. In most cases (but not in all) there will be at least one data directory in a section, with a pointer to it in the optional header's data directory array.

code section

First, I will mention the code section. The section will have, at least, the bits 'IMAGE_SCN_CNT_CODE', 'IMAGE_SCN_MEM_EXECUTE' and 'IMAGE_SCN_MEM_READ' set, and 'AddressOfEntryPoint' will point somewhere into the section, to the start of the function that the developer wants to execute first. 'BaseOfCode' will normally point to the start of this section, but may point to somewhere later in the section if some non-code-bytes are placed before the code in the section. Normally, there will be nothing but executable code in this section, and there will be only one code section, but don't rely on this. Typical section names are ".text", ".code", "AUTO" and the like.

data section

The next thing we'll discuss is the initialized variables; this section contains initialized static variables (like "static int i = 5;"). It will have, at least, the bits 'IMAGE_SCN_CNT_INITIALIZED_DATA', 'IMAGE_SCN_MEM_READ' and 'IMAGE_SCN_MEM_WRITE' set. Some linkers may place constant data into a section of their own that doesn't have the writeable-bit. If part of the data is shareable, or there are other peculiarities, there may be more sections with the apropriate section- bits set. The section, or sections, will be in the range 'BaseOfData' up to 'BaseOfData'+'SizeOfInitializedData'. Typical section names are '.data', '.idata', 'DATA' and so on.

bss section

Then there is the uninitialized data (for static variables like "static int k;"); this section is quite like the initialized data, but will have a file offset ('PointerToRawData') of 0 indicating its contents is not stored in the file, and 'IMAGE_SCN_CNT_UNINITIALIZED_DATA' is set instead of 'IMAGE_SCN_CNT_INITIALIZED_DATA' to indicate that the contents should be set to 0-bytes at load-time. This means, there is a section header but no section in the file; the section will be created by the loader and consist entirely of 0-bytes. The length will be 'SizeOfUninitializedData'. Typical names are '.bss', 'BSS' and the like.

These were the section data that are *not* pointed to by data directories. Their contents and structure is supplied by the compiler, not by the linker. (The stack-segment and heap-segment are not sections in the binary but created by the loader from the stacksize- and heapsize-entries in the optional header.)

To begin with a simple directory-section, let's look at the data directory 'IMAGE_DIRECTORY_ENTRY_COPYRIGHT'. The contents is a copyright- or description string in ASCII (not 0-terminated), like "Gonkulator control application, copyright (c) 1848 Hugendubel & Cie". This string is, normally, supplied to the linker with the command line or a description file. This string is not needed at runtime and may be discarded. It is not writeable; in fact, the application doesn't need access at all. So the linker will find out if there is a discardable non-writeable section already and if not, create one (named '.descr' or the like). It will then stuff the string into the section and let the copyright-directory-pointer point to the string. The 'IMAGE_SCN_CNT_INITIALIZED_DATA' bit should be set.

exported symbols

(Note that the description of the export directory was faulty in versions of this text before 1999-03-12. It didn't describe forwarders, exports by ordinal only, or exports with several names.)

The next-simplest thing is the export directory, 'IMAGE_DIRECTORY_ENTRY_EXPORT'. This is a directory typically found in DLLs; it contains the entry points of exported functions (and the addresses of exported objects etc.). Executables may of course also have exported symbols but usually they don't. The containing section should be "initialized data" and "readable". It should not be "discardable" because the process might call "GetProcAddress()" to find a function's entry point at runtime. The section is normally called '.edata' if it is a separate thing; often enough, it is merged into some other section like "initialized data".

The structure of the export table ('IMAGE_EXPORT_DIRECTORY') comprises a header and the export data, that is: the symbol names, their ordinals and the offsets to their entry points.

First, we have 32 bits of 'Characteristics' that are unused and normally 0. Then there is a 32-bit-'TimeDateStamp', which presumably should give the time the table was created in the time_t-format; alas, it is not always valid (some linkers set it to 0). Then we have 2 16-bit-words of version-info ('MajorVersion' and 'MinorVersion'), and these, too, are often enough set to 0.

The next thing is 32 bits of 'Name'; this is an RVA to the DLL name as a 0-terminated ASCII string. (The name is necessary in case the DLL file is renamed - see "binding" at the import directory.) Then, we have got a 32-bit-'Base'. We'll come to that in a moment.

The next 32-bit-value is the total number of exported items ('NumberOfFunctions'). In addition to their ordinal number, items may be exported by one or several names. and the next 32-bit-number is the total number of exported names ('NumberOfNames'). In most cases, each exported item will have exactly one corresponding name and it will be used by that name, but an item may have several associated names (it is then accessible by each of them), or it may have no name, in which case it is only accessible by its ordinal number. The use of unnamed exports (purely by ordinal) is discouraged, because all versions of the exporting DLL would have to use the same ordinal numbering, which is a maintainance problem.

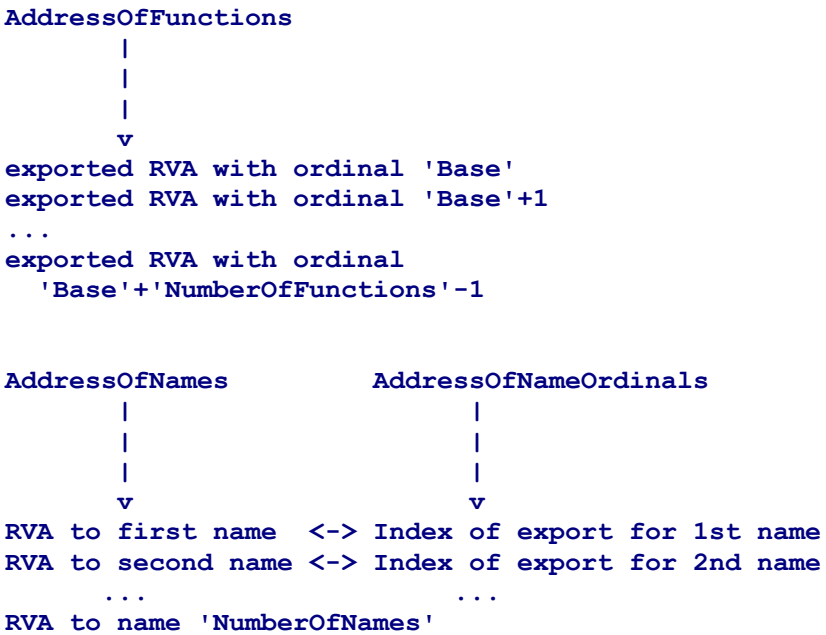
The next 32-bit-value 'AddressOfFunctions' is a RVA to the list of exported items. It points to an array of 'NumberOfFunctions' 32-bit-values, each being a RVA to the exported function or variable.

There are 2 quirks about this list: First, such an exported RVA may be 0, in which case it is unused. Second, if the RVA points into the section containing the export directory, this is a forwarded export. A forwarded export is a pointer to an export in another binary; if it is used, the pointed-to export in the other binary is used instead. The RVA in this case points, as mentioned, into the export directory's section, to a zero-terminated string comprising the name of the pointed-to DLL and the export name separated by a dot, like "otherdll.exportname", or the DLL's name and the export ordinal, like "otherdll.#19".

Now is the time to explain the export ordinal. An export's ordinal is the index into the AddressOfFunctions-Array (the 0-based position in this array) plus the 'Base' mentioned above. In most cases, the 'Base' is 1, which means the first export has an ordinal of 1, the second has an ordinal of 2 and so on.

After the 'AddressOfFunctions'-RVA we find a RVA to the array of 32-bit-RVAs to symbol names 'AddressOfNames', and a RVA to the array of 16-bit-ordinals 'AddressOfNameOrdinals'. Both arrays have 'NumberOfNames' elements. The symbol names may be missing entirely, in which case the 'AddressOfNames' is 0. Otherwise, the pointed-to arrays are running parallel, which means their elements at each index belong together. The 'AddressOfNames'-array consists of RVAs to 0-terminated export names; the names are held in a sorted list (i.e. the first array member is the RVA to the alphabetically smallest name; this allows efficient searching when looking up an exported symbol by name). According to the PE specification, the 'AddressOfNameOrdinals'-array has the ordinal corresponding to each name; however, I've found this array to contain the actual index into the 'AddressOfFunctions-Array instead.

I'll draw a picture about the three tables:



<-> Index of export for name
'NumberOfNames'

Some examples are in order.

To find an exported symbol by ordinal, subtract the 'Base' to get the index, follow the 'AddressOfFunctions'-RVA to find the exports-array and use the index to find the exported RVA in the array. If it does not point into the export section, you are done. Otherwise, it points to a string describing the exporting DLL and the name or ordinal therein, and you have to look up the forwarded export there.

To find an exported symbol by name, follow the 'AddressOfNames'-RVA (if it is 0 there are no names) to find the array of RVAs to the export names. Search your name in the list. Use the name's index in the 'AddressOfNameOrdinals'-Array and get the 16-bit-number corresponding to the found name. According to the PE spec, it is an ordinal and you need to subtract the 'Base' to get the export index; according to my experiences it is the export index and you don't subtract. Using the export index, you find the export RVA in the 'AddressOfFunctions'-Array, being either the exported RVA itself or a RVA to a string describing a forwarded export.

imported symbols

When the compiler finds a call to a function that is in a different executable (mostly in a DLL), it will, in the most simplistic case, not know anything about the circumstances and simply output a normal call-instruction to that symbol, the address of which the linker will have to fix, like it does for any external symbol. The linker uses an import library to look up from which DLL which symnol is imported, and produces stubs for all the imported symbols, each of which consists of a jump-instruction; the stubs are the actual call-targets. These jump-instructions will actually jump to an address that's fetched from the so-called import address table. In more sophisticated applications (when "__declspec(dllimport)" is used), the compiler knows the function is imported, and outputs a call to the address that's in the import address table, bypassing the jump.

Anyway, the address of the function in the DLL is always necessary and will be supplied by the loader from the exporting DLL's export directory when the application is loaded. The loader knows which symbols in what libraries have to be looked up and their addresses fixed by searching the import directory.

I will better give you an example. The calls with or without __declspec(dllimport) look like this:

source:

```
int symbol(char *);
__declspec(dllimport) int symbol2(char*);
void foo(void)
{
    int i=symbol("bar");
    int j=symbol2("baz");
}
```

assembly:

```
...
call _symbol  ; without declspec(dllimport)
...
call [__imp__symbol2] ; with declspec(dllimport)
...
```

In the first case (without __declspec(dllimport)), the compiler didn't know that '_symbol' was in a DLL, so the linker has to provide the function '_symbol'. Since the function isn't there, it will supply a stub function for the imported symbol, being an indirect jump. The collection of all import-stubs is called the "transfer area" (also sometimes called a "trampoline", because you jump there in order to jump to somewhere else). Typically this transfer area is located in the code section (it is not part of the import directory). Each of the function stubs is a jump to the actual function in the target DLLs. The transfer area looks like this:

```
_symbol:        jmp  [__imp__symbol]
_other_symbol:  jmp  [__imp__other__symbol]
...
```

This means: if you use imported symbols without specifying "__declspec(dllimport)" then the linker will generate a transfer area for them, consisting of indirect jumps. If you do specify "__declspec(dllimport)", the compiler will do the indirection itself and a transfer area is not necessary. (It also means: if you import variables or other stuff you must specify "__declspec(dllimport)", because a stub with a jmp instruction is appropriate for functions only.)

In any case the adress of symbol 'x' is stored at a location '__imp_x'. All these locations together comprise the so-called "import address table", which is provided to the linker by the import libraries of the various DLLs that are used. The import address table is a list of addresses like this:

```
__imp__symbol:   0xdeadbeef
__imp__symbol2:  0x40100
__imp__symbol3:  0x300100
...
```

This import address table is a part of the import directory, and it is pointed to by the IMAGE_DIRECTORY_ENTRY_IAT directory pointer (although some linkers don't set this directory entry and it works nevertheless; apparently, the loader can resolve imports without using the directory IMAGE_DIRECTORY_ENTRY_IAT). The addresses in this table are unknown to the linker; the linker inserts dummies (RVAs to the function names; see below for more information) that are patched by the loader at load time using the export directory of the exporting DLL. The import address table, and how it is found by the loader, will be described in more detail later in this chapter.

Note that this description is C-specific; there are other application building environments that don't use import libraries. They all need to generate an import address table, though, which they use to let their programs access the imported objects and functions. C compilers tend to use import libraries because it is convenient for them - their linkers use libraries anyway. Other environments use e.g. a description file that lists the necessary DLL names and function names (like the "module definition

file"), or a declaration-style list in the source.

This is how imports are used by the program's code; now we'll look how an import directory is made up so the loader can use it.

The import directory should reside in a section that's "initialized data" and "readable". The import directory is an array of IMAGE_IMPORT_DESCRIPTORs, one for each used DLL. The list is terminated by a IMAGE_IMPORT_DESCRIPTOR that's entirely filled with 0-bytes. An IMAGE_IMPORT_DESCRIPTOR is a struct with these members:

OriginalFirstThunk
An RVA (32 bit) pointing to a 0-terminated array of RVAs to IMAGE_THUNK_DATAs, each describing one imported function. The array will never change.

TimeDateStamp
A 32-bit-timestamp that has several purposes. Let's pretend that the timestamp is 0, and handle the advanced cases later.

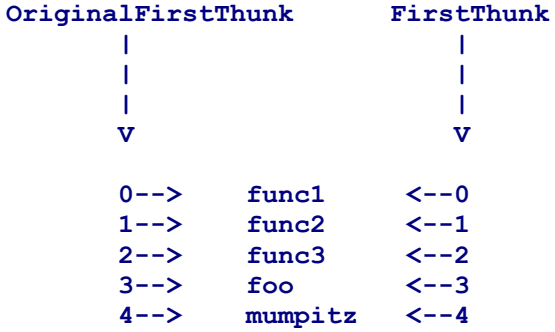
ForwarderChain
The 32-bit-index of the first forwarder in the list of imported functions. Forwarders are also advanced stuff; set to all-bits-1 for beginners.

Name
A 32-bit-RVA to the name (a 0-terminated ASCII string) of the DLL.

FirstThunk
An RVA (32 bit) to a 0-terminated array of RVAs to IMAGE_THUNK_DATAs, each describing one imported function. The array is part of the import address table and will change.

So each IMAGE_IMPORT_DESCRIPTOR in the array gives you the name of the exporting DLL and, apart from the forwarder and timestamp, it gives you 2 RVAs to arrays of IMAGE_THUNK_DATAs, using 32 bits. (The last member of each array is entirely filled with 0-bytes to mark the end.) Each IMAGE_THUNK_DATA is, for now, an RVA to a IMAGE_IMPORT_BY_NAME which describes the imported function. The interesting point is now, the arrays run parallel, i.e.: they point to the same IMAGE_IMPORT_BY_NAMEs.

No need to be desparate, I will draw another picture. This is the essential contents of one IMAGE_IMPORT_DESCRIPTOR:



```
            5-->    knuff     <--5
            6-->0           0<--6  /* the last RVA is 0! */
```

where the names in the center are the yet to discuss IMAGE_IMPORT_BY_NAMEs. Each of them is a 16-bit-number (a hint) followed by an unspecified amount of bytes, being the 0-terminated ASCII name of the imported symbol. The hint is an index into the exporting DLL's name table (see export directory above). The name at that index is tried, and if it doesn't match then a binary search is done to find the name. (Some linkers don't bother to look up correct hints and simply specify 1 all the time, or some other arbitrary number. This doesn't harm, it just makes the first attempt to resolve the name always fail, enforcing a binary search for each name.)

To summarize, if you want to look up information about the imported function "foo" from DLL "knurr", you first find the entry IMAGE_DIRECTORY_ENTRY_IMPORT in the data directories, get an RVA, find that address in the raw section data and now have an array of IMAGE_IMPORT_DESCRIPTORs. Get the member of this array that relates to the DLL "knurr" by inspecting the strings pointed to by the 'Name's. When you have found the right IMAGE_IMPORT_DESCRIPTOR, follow its 'OriginalFirstThunk' and get hold of the pointed-to array of IMAGE_THUNK_DATAs; inspect the RVAs and find the function "foo".

Ok, now, why do we have *two* lists of pointers to the IMAGE_IMPORT_BY_NAMEs? Because at runtime the application doesn't need the imported functions' names but the addresses. This is where the import address table comes in again. The loader will look up each imported symbol in the export-directory of the DLL in question and replace the IMAGE_THUNK_DATA-element in the 'FirstThunk'-list (which until now also points to the IMAGE_IMPORT_BY_NAME) with the linear address of the DLL's entry point. Remember the list of addresses with labels like "__imp__symbol"; the import address table, pointed to by the data directory IMAGE_DIRECTORY_ENTRY_IAT, is exactly the list pointed to by 'FirstThunk'. (In case of imports from several DLLs, the import address table comprises the 'FirstThunk'-Arrays of all the DLLs. The directory entry IMAGE_DIRECTORY_ENTRY_IAT may be missing, the imports will still work fine.) The 'OriginalFirstThunk'-array remains untouched, so you can always look up the original list of imported names via the 'OriginalFirstThunk'-list.

The import is now patched with the correct linear addresses and looks like this:

```
  OriginalFirstThunk      FirstThunk
         |                    |
         |                    |
         |                    |
         V                    V

         0-->    func1        0-->  exported func1
         1-->    func2        1-->  exported func2
         2-->    func3        2-->  exported func3
         3-->    foo          3-->  exported foo
         4-->    mumpitz      4-->  exported mumpitz
         5-->    knuff        5-->  exported knuff
         6-->0           0<--6
```

This was the basic structure, for simple cases. Now we'll learn about tweaks in the import directories.

First, the bit IMAGE_ORDINAL_FLAG (that is: the MSB) of the IMAGE_THUNK_DATA in the arrays can be set, in which case there is no symbol-name-information in the list and the symbol is imported purely by ordinal. You get the ordinal by inspecting the lower word of the IMAGE_THUNK_DATA. The import by ordinals is discouraged; it is much safer to import by name, because the export ordinals might change if the exporting DLL is not in the expected version.

Second, there are the so-called "bound imports".

Think about the loader's task: when a binary that it wants to execute needs a function from a DLL, the loader loads the DLL, finds its export directory, looks up the function's RVA and calculates the function's entry point. Then it patches the so-found address into the 'FirstThunk'- list. Given that the programmer was clever and supplied unique preferred load addresses for the DLLs that don't clash, we can assume that the functions' entry points will always be the same. They can be computed and patched into the 'FirstThunk'-list at link-time, and that's what happens with the "bound imports". (The utility "bind" does this; it is part of the Win32 SDK.)

Of course, one must be cautious: The user's DLL may have a different version, or it may be necessary to relocate the DLL, thus invalidating the pre-patched 'FirstThunk'-list; in this case, the loader will still be able to walk the 'OriginalFirstThunk'-list, find the imported symbols and re-patch the 'FirstThunk'-list. The loader knows that this is necessary if a) the versions of the exporting DLL don't match or b) the exporting DLL had to be relocated.

To decide whether there were relocations is no problem for the loader, but how to find out if the versions differ? This is where the 'TimeDateStamp' of the IMAGE_IMPORT_DESCRIPTOR comes in. If it is 0, the import-list has not been bound, and the loader must fix the entry points always. Otherwise, the imports are bound, and 'TimeDateStamp' must match the 'TimeDateStamp' of the exporting DLL's 'FileHeader'; if it doesn't match, the loader assumes that the binary is bound to a "wrong" DLL and will re-patch the import list.

There is an additional quirk about "forwarders" in the import-list. A DLL can export a symbol that's not defined in the DLL but imported from another DLL; such a symbol is said to be forwarded (see the export directory description above). Now, obviously you can't tell if the symbol's entry point is valid by looking into the timestamp of a DLL that doesn't actually contain the entry point. So the forwarded symbols' entry points must always be fixed up, for safety reasons. In the import list of a binary, imports of forwarded symbols need to be found so the loader can patch them.

This is done via the 'ForwarderChain'. It is an index into the thunk- lists; the import at the indexed position is a forwarded export, and the contents of the 'FirstThunk'-list at this position is the index of the *next* forwarded import, and so on, until the index is "-1" which indicates there are no more forwards. If there are no forwarders at all, 'ForwarderChain' is -1 itself.

This was the so-called "old-style" binding.

At this point, we should sum up what we have had so far :-)

Ok, I will assume you have found the IMAGE_DIRECTORY_ENTRY_IMPORT and you have followed it to find the import-directory, which will be in one of the sections. Now you're at the beginning of an array of IMAGE_IMPORT_DESCRIPTORs the last of which will be entirely 0-bytes- filled. To decipher one of the

IMAGE_IMPORT_DESCRIPTORs, you first look into the 'Name'-field, follow the RVA and thusly find the name of the exporting DLL. Next you decide whether the imports are bound or not; 'TimeDateStamp' will be non-zero if the imports are bound. If they are bound, now is a good time to check if the DLL version matches yours by comparing the 'TimeDateStamp's. Now you follow the 'OriginalFirstThunk'-RVA to go to the IMAGE_THUNK_DATA-array; walk down this array (it is be 0-terminated), and each member will be the RVA of a IMAGE_IMPORT_BY_NAME (unless the hi-bit is set in which case you don't have a name but are left with a mere ordinal). Follow the RVA, and skip 2 bytes (the hint), and now you have got a 0-terminated ASCII-string that's the name of the imported function. To find the supplied entry point addresses in case it is a bound import, follow the 'FirstThunk' and walk it parallel to the 'OriginalFirstThunk'-array; the array-members are the linear addresses of the entry points (leaving aside the forwarders-topic for a moment).

There is one thing I didn't mention until now: Apparently there are linkers that exhibit a bug when they build the import directory (I've found this bug being in use by a Borland C linker). These linkers set the 'OriginalFirstThunk' in the IMAGE_IMPORT_DESCRIPTOR to 0 and create only the 'FirstThunk'-array. Obviously, such import directories cannot be bound (else the necessary information to re-fix the imports were lost - you couldn't find the function names). In this case, you will have to follow the 'FirstThunk'-array to get the imported symbol names, and you will never have pre-patched entry point addresses. I have found a TIS document ([6]) describing the import directory in a way that is compatible to this bug, so that paper may be the origin of the bug.

The TIS document specifies:

```
    IMPORT FLAGS
    TIME/DATE STAMP
    MAJOR VERSION - MINOR VERSION
    NAME RVA
    IMPORT LOOKUP TABLE RVA
    IMPORT ADDRESS TABLE RVA
```

as opposed to the structure used elsewhere:

```
    OriginalFirstThunk
    TimeDateStamp
    ForwarderChain
    Name
    FirstThunk
```

The last tweak about the import directories is the so-called "new style" binding (it is described in [3]), which can also be done with the "bind"-utility. When this is used, the 'TimeDateStamp' is set to all-bits-1 and there is no forwarderchain; all imported symbols get their address patched, whether they are forwarded or not. Still, you need to know the DLLs' version, and you need to distinguish forwarded symbols from ordinary ones. For this purpose, the IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT directory is created. This will, as far as I could find out, *not* be in a section but in the header, after the section headers and before the first section. (Hey, I didn't invent this, I'm only describing it!) This directory tells you, for each used DLL, from which other DLLs there are forwarded exports. The structure is an IMAGE_BOUND_IMPORT_DESCRIPTOR, comprising (in this order): A 32-bit number, giving you the 'TimeDateStamp' of the DLL; a 16-bit-number 'OffsetModuleName', being the offset from the beginning of the directory to the 0-terminated name of the DLL; a 16-bit-number

'NumberOfModuleForwarderRefs' giving you the number of DLLs that this DLL uses for its forwarders.

Immediately following this struct you find 'NumberOfModuleForwarderRefs' structs that tell you the names and versions of the DLLs that this DLL forwards from. These structs are 'IMAGE_BOUND_FORWARDER_REF's: A 32-bit-number 'TimeDateStamp'; a 16-bit-number 'OffsetModuleName', being the offset from the beginning of the directory to the 0-terminated name of the forwarded-from DLL; 16 unused bits.

Following the 'IMAGE_BOUND_FORWARDER_REF's is the next 'IMAGE_BOUND_IMPORT_DESCRIPTOR' and so on; the list is terminated by an all-0-bits-IMAGE_BOUND_IMPORT_DESCRIPTOR.

Sorry for the inconvenience, but that's what it looks like :-)

Now, if you have a new-bound import directory, you load all the DLLs, use the directory pointer IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT to find the IMAGE_BOUND_IMPORT_DESCRIPTOR, scan through it and check if the 'TimeDateStamp's of the loaded DLLs match the ones given in this directory. If not, fix them in the 'FirstThunk'-array of the import directory.

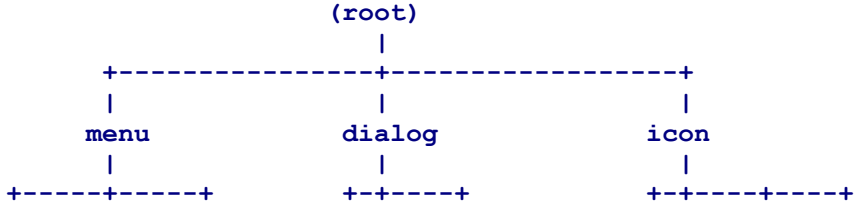
resources

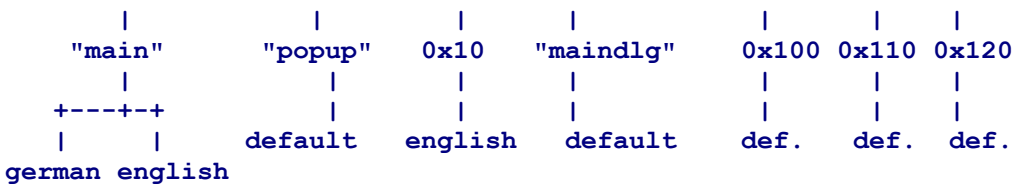
The resources, such as dialog boxes, menus, icons and so on, are stored in the data directory pointed to by IMAGE_DIRECTORY_ENTRY_RESOURCE. It is in a section that has, at least, the bits 'IMAGE_SCN_CNT_INITIALIZED_DATA' and 'IMAGE_SCN_MEM_READ' set.

A resource base is a 'IMAGE_RESOURCE_DIRECTORY'; it contains several 'IMAGE_RESOURCE_DIRECTORY_ENTRY's each of which in turn may point to a 'IMAGE_RESOURCE_DIRECTORY'. This way, you get a tree of 'IMAGE_RESOURCE_DIRECTORY's with 'IMAGE_RESOURCE_DIRECTORY_ENTRY's as leafs; these leafs point to the actual resource data.

In real life, the situation is somewhat relaxed. Normally you won't find convoluted trees you can't possibly sort out. The hierarchy is, normally, like this: one directory is the root. It points to directories, one for each resource type. These directories point to subdirectories, each of which will have a name or an ID and point to a directory of the languages provided for this resource; for each language you will find one resource entry, which will finally point to the data. (Note that multi-language-resources don't work on Win95, which always uses the same resource if it is available in several languages - I didn't check which one, but I guess it's the first it encounters. They do work on NT.)

The tree, without the pointer to the data, may look like this:





A IMAGE_RESOURCE_DIRECTORY comprises:
32 bits of unused flags called 'Characteristics';
32 bits 'TimeDateStamp' (again in the common time_t representation), giving you the time the resource was created (if the entry is set); 16 bits 'MajorVersion' and 16 bits 'MinorVersion', thusly allowing you to maintain several versions of the resource; 16 bits 'NumberOfNamedEntries' and another 16 bits 'NumberOfIdEntries'.

Immediately following such a structure are 'NumberOfNamedEntries'+'NumberOfIdEntries' structs which are of the format 'IMAGE_RESOURCE_DIRECTORY_ENTRY', those with the names coming first. They may point to further 'IMAGE_RESOURCE_DIRECTORY's or they point to the actual resource data. A IMAGE_RESOURCE_DIRECTORY_ENTRY consists of: 32 bits giving you the id of the resource or the directory it describes; 32 bits offset to the data or offset to the next sub-directory.

The meaning of the id depends on the level in the tree; the id may be a number (if the hi-bit is clear) or a name (if the hi-bit is set). If it is a name, the lower 31 bits are the offset from the beginning of the resource section's raw data to the name (the name consists of 16 bits length and trailing wide characters, in unicode, not 0-terminated).

If you are in the root-directory, the id, if it is a number, is the resource-type:
    1: cursor
    2: bitmap
    3: icon
    4: menu
    5: dialog
    6: string table
    7: font directory
    8: font
    9: accelerators
    10: unformatted resource data
    11: message table
    12: group cursor
    14: group icon
    16: version information
Any other number is user-defined. Any resource-type with a type-name is always user-defined.

If you are one level deeper, the id is the resource-id (or resource- name).

If you are another level deeper, the id must be a number, and it is the language-id of the specific instance of the resource; for example, you can have the same dialog in australian english, canadian french and swiss german localized forms, and they all share the same resource-id. The system will choose the dialog to load based on the thread's locale, which in turn will usually reflect the user's "regional setting". (If the resource cannot be

found for the thread locale, the system will first try to find a resource for the locale using a neutral sublanguage, e.g. it will look for standard french instead of the user's canadian french; if it still can't be found, the instance with the smallest language id will be used. As noted, all this works only on NT.) To decipher the language id, split it into the primary language id and the sublanguage id using the macros PRIMARYLANGID() and SUBLANGID(), giving you the bits 0 to 9 or 10 to 15, respectivly. The values are defined in the file "winresrc.h". Language-resources are only supported for accelerators, dialogs, menus, rcdata or stringtables; other resource-types should be LANG_NEUTRAL/SUBLANG_NEUTRAL.

To find out whether the next level below a resource directory is another directory, you inspect the hi-bit of the offset. If it is set, the remaining 31 bits are the offset from the beginning of the resource section's raw data to the next directory, again in the format IMAGE_RESOURCE_DIRECTORY with trailing IMAGE_RESOURCE_DIRECTORY_ENTRYs.

If the bit is clear, the offset is the distance from the beginning of the resource section's raw data to the resource's raw data description, a IMAGE_RESOURCE_DATA_ENTRY. It consists of 32 bits 'OffsetToData' (the offset to the raw data, counting from the beginning of the resource section's raw data), 32 bits of 'Size' of the data, 32 bits 'CodePage' and 32 unused bits. (The use of codepages is discouraged, you should use the 'language'- feature to support multiple locales.)

The raw data format depends on the resource type; descriptions can be found in the MS SDK documentation. Note that any string in resources is always in UNICODE except for user defined resources, which are in the format the developer chooses, obviously.

relocations

The last data directory I will describe is the base relocation directory. It is pointed to by the IMAGE_DIRECTORY_ENTRY_BASERELOC entry in the data directories of the optional header. It is typically contained in a section if its own, with a name like ".reloc" and the bits IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_DISCARDABLE and IMAGE_SCN_MEM_READ set.

The relocation data is needed by the loader if the image cannot be loaded to the preferred load address 'ImageBase' mentioned in the optional header. In this case, the fixed addresses supplied by the linker are no longer valid, and the loader has to apply fixups for absolute addresses used for locations of static variables, string literals and so on.

The relocation directory is a sequence of chunks. Each chunk contains the relocation information for 4 KB of the image. A chunk starts with a 'IMAGE_BASE_RELOCATION' struct. It consists of 32 bits 'VirtualAddress' and 32 bits 'SizeOfBlock'. It is followed by the chunk's actual relocation data, being 16 bits each. The 'VirtualAddress' is the base RVA that the relocations of this chunk need to be applied to; the 'SizeOfBlock' is the size of the entire chunk in bytes. The number of trailing relocations is ('SizeOfBlock'-sizeof(IMAGE_BASE_RELOCATION))/2 The relocation information ends when you encounter a IMAGE_BASE_RELOCATION struct with a 'VirtualAddress' of 0.

Each 16-bit-relocation information consists of the relocation position in the

lower 12 bits and a relocation type in the high 4 bits. To get the relocation RVA, you need to add the IMAGE_BASE_RELOCATION's 'VirtualAddress' to the 12-bit-position. The type is one of:

IMAGE_REL_BASED_ABSOLUTE (0)
This is a no-op; it is used to align the chunk to a 32-bits-border. The position should be 0.

IMAGE_REL_BASED_HIGH (1)
The high 16 bits of the relocation must be applied to the 16 bits of the WORD pointed to by the offset, which is the high word of a 32-bit-DWORD.

IMAGE_REL_BASED_LOW (2)
The low 16 bits of the relocation must be applied to the 16 bits of the WORD pointed to by the offset, which is the low word of a 32-bit-DWORD.

IMAGE_REL_BASED_HIGHLOW (3)
The entire 32-bit-relocation must be applied to the entire 32 bits in question. This (and the no-op '0') is the only relocation type I've actually found in binaries.

IMAGE_REL_BASED_HIGHADJ (4)
This is one for the tough. Read yourself (from [6]) and make sense out of it if you can:
"Highadjust. This fixup requires a full 32-bit value. The high 16-bits is located at Offset, and the low 16-bits is located in the next Offset array element (this array element is included in the Size field). The two need to be combined into a signed variable. Add the 32-bit delta. Then add 0x8000 and store the high 16-bits of the signed variable to the 16-bit field at Offset."

IMAGE_REL_BASED_MIPS_JMPADDR (5)
Unknown

IMAGE_REL_BASED_SECTION (6)
Unknown

IMAGE_REL_BASED_REL32 (7)
Unknown

As an example, if you find the relocation information to be

```
    0x00004000      (32 bits, starting RVA)
    0x00000010      (32 bits, size of chunk)
    0x3012          (16 bits reloc data)
    0x3080          (16 bits reloc data)
    0x30f6          (16 bits reloc data)
    0x0000          (16 bits reloc data)
    0x00000000      (next chunk's RVA)
    0xff341234
```

you know the first chunk describes relocations starting at RVA 0x4000 and is 16 bytes long. Because the header uses 8 bytes and one relocation uses 2 bytes, there are (16-8)/2=4 relocations in the chunk. The first relocation is to be applied to the DWORD at 0x4012, the next to the DWORD at 0x4080, and the third to the DWORD at 0x40f6. The last relocation is a no-op. The next chunk has a RVA of 0 and finishes the list.

Now, how do you do a relocation? You know that the image *is* relocated to

the preferred load address 'ImageBase' in the optional header; you also know the address you did load the image to. If they match, you don't need to do anything. If they don't match, you calculate the difference actual_base-preferred_base and add that value (signed, it may be negative) to the relocation positions, which you will find with the method described above.

Acknowledgments

Thanks go to David Binette for his debugging and proof-reading.
(The remaining errors are entirely mine.)
Also thanks to wotsit.org for letting me put the file on their site.

Copyright

Literature

[1]
"Peering Inside the PE: A Tour of the Win32 Portable Executable File Format" (M. Pietrek), in: Microsoft Systems Journal 3/1994

[2]
"Why to Use _declspec(dllimport) & _declspec(dllexport) In Code", MS Knowledge Base Q132044

[3]
"Windows Q&A" (M. Pietrek), in: Microsoft Systems Journal 8/1995

[4]
"Writing Multiple-Language Resources", MS Knowledge Base Q89866

[5]
"The Portable Executable File Format from Top to Bottom" (Randy Kath), in: Microsoft Developer Network

[6]
Tool Interface Standard (TIS) Formats Specification for Windows Version 1.0 (Intel Order Number 241597, Intel Corporation 1993)