Wine Documentation

Wine Documentation

Table of Contents

Notes About this Document....11
I. Using Wine....12
1. Running Wine....13
1.1. How to run Wine....13
2. Installing Wine....14
2.1. WWN #52 Feature: Replacing Windows....14
2.1.1. Installation Overview....14
2.1.2. The Registry....14
2.1.3. Directory Structure....15
2.1.4. System DLLs....16
2.2. Installing Wine Without Windows....17
2.3. Dealing With FAT/VFAT Partitions....19
2.3.1. Introduction....19
2.3.2. Running Wine as root....20
2.3.3. Mounting FAT filesystems....20
2.3.4. Shadowing FAT filesystems....22
2.4. SCSI Support....23
2.4.1. Windows requirements....24
2.4.2. LINUX requirements:....24
2.4.3. General Information....25
2.4.4. NOTES/BUGS....25
3. Configuring Wine....27
3.1. General Configuration....27
3.1.1. The Wine Config File....27
3.1.2. How Do I Make One?....27
3.1.2.1. The [Drive X] Section....28
3.1.2.2. The [wine] Section....31
3.1.2.3. Introduction To DLL Sections....32
3.1.2.3.1. Windows DLL Pairs....32
3.1.2.3.2. Different Forms Of DLL's....33
3.1.2.4. The [DllDefaults] Section....34

3.1.2.5. The [DllPairs] Section....35
3.1.2.6. The [DllOverrides] Section....35
3.1.2.7. The [options] Section....37
3.1.2.8. The [fonts] Section....37
3.1.2.9. The [serialports], [parallelports], [spooler], and [ports] Sections....38
3.1.2.10. The [spy], [Registry], [tweak.layout], and [programs] Sections....40
3.1.3. Where Do I Put It?....41
3.1.4. What If It Doesn't Work?....42
3.2. Win95/98 Look....42
3.3. Configuring the x11drv Driver....43
3.3.1. x11drv modes of operation....44
3.3.2. The [x11drv] section....45
3.4. The Registry....47
3.4.1. Registry structure....47
3.4.2. Using a Windows registry....48
3.4.3. Wine registry data filesr....48
3.4.4. System administration....49
3.4.5. The default registry....50
3.4.6. The [registry] section....51
3.5. Drive labels and serial numbers with wine....52
3.5.1. What's Supported?....52
3.5.2. How To Set Up?....53
3.5.3. EXAMPLES....53
3.5.4. Todo / Open Issues....54
3.6. Dll Overrides....54
3.6.1. DLL types....55
3.6.2. The [DllDefaults] section....55
3.6.3. The [DllPairs] section....56
3.6.4. The [DllOverrides] section....56
3.7. Keyboard....61
4. Finding and Reporting Bugs....65
4.1. How To Report A Bug....65

4.1.1. The Easy Way 65
4.1.2. The Hard Way 65
4.1.3. Questions and comments 67
5. Dealing with Fonts 69
5.1. Fonts 69
5.1.1. How To Convert Windows Fonts 69
5.1.2. How To Add Font Aliases To `wine.conf` 70
5.1.3. How To Manage Cached Font Metrics 73
5.1.4. Too Small Or Too Large Fonts 73
5.1.5. "FONT_Init: failed to load ..." Messages On Startup 73
5.2. Setting up a TrueType Font Server 74
6. Printing in Wine 77
6.1. Printing 77
6.1.1. External printer drivers 77
6.1.2. Builtin Wine PostScript driver 78
6.1.3. Spooling 78
6.2. The Wine PostScript Driver 78
6.2.1. Installation 78
6.2.2. TODO / Bugs 80
II. Developing Wine 82
7. Compiling Wine 83
8. Debugging Wine 84
8.1. Debug Messages 84
8.1.1. Debugging classes 84
8.1.2. Debugging channels 84
8.1.3. How to use it 85
8.1.4. Are we debugging? 85
8.1.5. In-memory messages 85
8.1.6. Resource identifiers 87
8.1.7. The *--debugmsg* command line option 87
8.1.8. Compiling Out Debugging Messages 88
8.1.9. A Few Notes on Style 88
8.2. Using the Wine Debugger 89

8.2.1. Crashes 89
8.2.2. Program hangs, nothing happens 90
8.2.3. Program reports an error with a Messagebox 90
8.2.4. Disassembling programs: 91
8.2.5. Sample debugging session: 91
8.2.6. Debugging Tips 93
8.2.7. Some basic debugger usages: 94
8.3. How to do regression testing using Cvs 94
9. Documenting Wine 97
9.1. Writing Wine API Documentation 97
10. Submitting Patches 99
11. Internationalization 100
11.1. Adding New Languages 100
12. Porting Wine to new Platforms 101
12.1. Porting 101
12.1.1. What is this? 101
12.1.2. Why #ifdef MyOS is probably a mistake. 101
12.1.3. MyOS doesn't have the `foo.h` header! 101
12.1.4. MyOS doesn't have the `bar` function! 101
12.2. Running & Compiling WINE in OS/2 102
III. Distributing Wine 104
13. Packaging Wine 105
13.1. A Small WINE Distribution Guide 105
IV. Wine Architecture 113
14. Overview 114
14.1. Basic Overview 114
14.1.1. Wine Overview 114
14.1.2. Win16 and Win32 114
14.1.3. The Wineserver 114
14.1.4. The Service Thread 114
14.1.5. Relays, Thunks, and DLL descriptors 115
14.1.6. Core and non-core DLLs 115
14.2. Module Overview 115

14.2.1. KERNEL Module 116
14.2.2. GDI Module 116
14.2.2.1. X Windows System interface 116
14.2.3. USER Module 116
14.2.3.1. Windowing subsystem 116
14.2.3.1.1. Visible region, clipping region and update region 117
14.2.3.2. Messaging subsystem 118
14.3. WINE/WINDOWS DLLs 119
14.3.1. Pros of Native DLLs 119
14.3.2. Cons of Native DLLs 120
14.3.3. Deciding Between Native and Built-In DLLs 120
14.3.4. Load Order for DLLs 120
14.3.5. Understanding What DLLs Do 120
15. The Wine Debugger 122
15.1. I Introduction 122
15.1.1. I.1 Processes and threads: in underlying OS and in Windows 122
15.1.2. I.2 Wine, debugging and WineDbg 122
15.2. II WineDbg's modes of invocation 122
15.2.1. II.1 Starting a process 122
15.2.2. II.2 Attaching 122
15.2.3. II.3 On exception 122
15.2.4. II.4 Quitting 123
15.3. III Configuration 123
15.3.1. III.1 Registry configuration 123
15.3.2. III.2 WineDbg configuration 123
15.3.2.1. III.2.1 Controling when the debugger is entered 124
15.3.2.2. III.2.2 Output handling 124
15.3.2.3. III.2.2 Context information 125
15.3.3. III.3 Finding files 125
15.4. IV WineDbg commands 125
15.4.1. IV.1 Misc 125
15.4.2. IV.2 Flow control 125
15.4.3. IV.3 Breakpoints, watch points 125

15.4.4. IV.4 Stack manipulation 126
15.4.5. IV.5 Directory & source file manipulation 126
15.4.6. IV.6 Displaying 126
15.4.7. IV.7 Disassembly 126
15.4.8. IV.8 Information on Wine's internals 127
15.4.9. IV.9 Memory (reading, writing, typing) 127
15.5. V Other debuggers 127
15.5.1. V.1 Using other Unix debuggers 127
15.5.2. V.2 Using other Windows debuggers 127
15.5.3. V.3 Main differences between winedbg and regular Unix debuggers 128
15.6. VI Limitations 128
16. Consoles in Wine 129
16.1. Consoles 129
16.1.1. Console - First Pass 129
16.1.2. BUGS 129
16.1.3. Experimentation 129
16.1.4. DOS (Generic) Console Support 130
16.1.4.1. I. Command Line Configuration 130
16.1.4.2. II. `wine.conf` Configuration 130
16.1.4.3. III. Terminal Types 130
17. Low-level Implementation 131
17.1. Builtin DLLs 131
17.1.1. 1. The LibMain function 131
17.1.2. 2. Using functions from other built-in DLL's 131
17.1.3. 3. Getting resources from a `*.rc` file linked to the DLL 132
17.2. Accelerators 132
17.3. File Handles 132
17.4. Doing A Hardware Trace In Wine 133
18. Wine and OpenGL 137
18.1. I What is needed to have OpenGL support in Wine 137
18.1.1. I.1 Header files 137
18.1.2. I.2 OpenGL library thread-safety 137
18.1.3. I.3 OpenGL library itself 137

18.1.4. I.4 glXGetProcAddressARB function 137
18.2. II How to configure 137
18.3. III How it all works 137
18.3.1. III.1 The Windowing system integration 138
18.3.2. III.2 The thunks 138
18.4. IV Known problems - shortcomings 138
18.4.1. IV.1 Missing GLU32.DLL 138
18.4.2. IV.2 OpenGL not detected at configure time 138
18.4.3. IV.3 When running an OpenGL application, the screen flickers 138
18.4.4. IV.4 Wine gives me the following error message : 139
18.4.5. IV.5 `libopengl32.so` is built but it is still not working 139
19. The Wine Build System 140
20. Tools 141
20.1. bin2res 141
20.1.1. bin2res 141
20.1.2. Create binary files from an `.rc` file 141
20.1.3. Create a `.rc` file from binaries 141
20.1.4. output of **bin2res** 141
21. Wine Builtin DLLs 142
21.1. Common Controls 142
21.1.1. 1. Introduction 142
21.1.2. 2. General Information 142
21.1.2.1. 2.1 Structure sizes of different common control versions 142
21.1.3. 3. Controls 142
21.1.3.1. 3.1 Animation Control 142
21.1.3.2. 3.2 Combo Box Ex Control 143
21.1.3.3. 3.3 Date and Time Picker Control 143
21.1.3.4. 3.4 Drag List Box Control 143
21.1.3.5. 3.5 Flat Scroll Bar Control 143
21.1.3.6. 3.6 Header Control 143
21.1.3.7. 3.7 Hot Key Control 143
21.1.3.8. 3.8 Image List (no control) 143

21.1.3.9. 3.9 IP Address Control....143
21.1.3.10. 3.10 List View Control....144
21.1.3.11. 3.11 Month Calendar Control....144
21.1.3.12. 3.12 Native font control....144
21.1.3.13. 3.13 Pager Control....144
21.1.3.14. 3.14 Progress Bar Control....144
21.1.3.15. 3.15 Property Sheet....144
21.1.3.16. 3.16 Rebar Control (Cool Bar)....144
21.1.3.17. 3.17 Status Bar Control....145
21.1.3.18. 3.18 Tab Control....145
21.1.3.19. 3.19 Toolbar Control....145
21.1.3.20. 3.20 Tooltip Control....145
21.1.3.21. 3.21 Trackbar Control....145
21.1.3.22. 3.22 Tree View Control....145
21.1.3.23. 3.23 Updown Control....145
21.1.4. 4. Additional Information....145
21.1.5. 5. Undocumented features....145
21.1.5.1. 5.1 Dymnamic Storage Array (DSA)....146
21.1.5.2. 5.2 Dynamic Pointer Array (DPA)....146
21.1.5.3. 5.3 "Most Recently Used" - List (MRU)....146
21.1.5.4. 5.4 MenuHelp....146
21.1.5.5. 5.5 GetEffectiveClientRect....146
21.1.5.6. 5.6 ShowHideMenuCtl....146
21.1.5.7. 5.7 Other undocumented functions....146
21.1.6. 6. Epilogue....146

Notes About this Document

The following documentation has been extracted from the Wine source tree, from the `wine/documentation` directory. So far, no new content has been added aside from marking it up for DocBook and a few minor tweaks to smooth that process. All credit should go to the original authors, who are attributed according the the "written by" comments in those files. If I've missed anyone, please contact <`John R. Sheets <jsheets@codeweavers.com>`>

I. Using Wine

Table of Contents

1. Running Wine 13
2. Installing Wine 14
3. Configuring Wine 27
4. Finding and Reporting Bugs 65
5. Dealing with Fonts 69
6. Printing in Wine 77

Chapter 1. Running Wine

Explain the command-line options you can use to run Wine

1.1. How to run Wine

The first thing you need to do is...

Chapter 2. Installing Wine

How to install Wine...

2.1. WWN #52 Feature: Replacing Windows

Written by Ove Kåven <ovek@winehq.com>

2.1.1. Installation Overview

A Windows installation consists of many different parts.

- Registry. Many keys are supposed to exist and contain meaningful data, even in a newly-installed Windows.
- Directory structure. Applications expect to find and/or install things in specific predetermined locations. Most of these directories are expected to exist. But unlike Unix directory structures, most of these locations are not hardcoded, and can be queried via the Windows API and the registry. This places additional requirements on a Wine installation.
- System DLLs. In Windows, these usually reside in the `system` (or `system32`) directories. Some Windows applications check for their existence in these directories before attempting to load them. While Wine is able to load its own internal DLLs (`.so` files) when the application asks for a DLL, Wine does not simulate the existence of nonexisting files.

While the users are of course free to set up everything themselves, the Wine team will make the automated Wine installation script, `tools/wineinstall`, do everything we find necessary to do; running the conventional **configure && make depend && make && make install** cycle is thus not recommended, unless you know what you're doing. At the moment, `tools/wineinstall` is able to create a configuration file, install the registry, and create the directory structure itself.

2.1.2. The Registry

The default registry is in the file `winedefault.reg`. It contains directory paths, class IDs, and more; it must be installed before most `INSTALL.EXE` or `SETUP.EXE` applications will work. The registry is covered in more detail in an earlier article.

2.1.3. Directory Structure

Here's the fundamental layout that Windows applications and installers expect. Without it, they seldom operate correctly.

C:\		Root directory of primary disk drive
	Windows\	Windows directory, containing .INI files, accessories, etc
	System\	Win3.x/95/98/ME directory for common DLLs WinNT/2000 directory for common 16-bit DLLs
	System32\	WinNT/2000 directory for common 32-bit DLLs

	Start Menu\		Program launcher directory structure
		Programs\	Program launcher links (.LNK files) to applications
Program Files\			Application binaries (.EXE and .DLL files)

Wine emulates drives by placing their virtual drive roots to user-configurable points in the Unix filesystem, so it's your choice where *C:*'s root should be (`tools/wineinstall` will even ask you). If you choose, say, `/var/wine`, as the root of your virtual drive *C*, then you'd put this in your `wine.conf`:

```
[Drive C]
Path=/var/wine
Type=hd
Label=MS-DOS
Filesystem=win95
```

With this configuration, what windows apps think of as "c:\windows\system" would map to `/var/wine/windows/system` in the UNIX filesystem. Note that you need to specify `Filesystem=win95`, NOT `Filesystem=unix`, to make Wine simulate a Windows-compatible (case-insensitive) filesystem, otherwise most apps won't work.

2.1.4. System DLLs

The Wine team has determined that it is necessary to create fake DLL files to trick many applications that check for file existence to determine whether a particular feature (such as Winsock and its TCP/IP networking) is available. If this is a problem for you, you can create empty files in the `system` directory to make the application think it's there, and Wine's built-in DLL will be loaded when the application actually asks for it. (Unfortunately, `tools/wineinstall` does not create such empty files itself.)

Applications sometimes also try to inspect the version resources from the physical files (for example, to determine the DirectX version). Empty files will not do in this case, it is rather necessary to install files with complete version resources. This problem is currently being worked on. In the meantime, you may still need to grab some real DLL files to fool these apps with.

And there are of course DLLs that wine does not currently implement very well (or at all). If you do not have a real Windows you can steal necessary DLLs from, you can always get some from a DLL archive such as http://solo.abac.com/dllarchive/.

2.2. Installing Wine Without Windows

written by ???

(Extracted from `wine/documentation/no-windows`)

A major goal of Wine is to allow users to run Windows programs without having to install Windows on their machine. Wine implements the functionality of the main DLL's usually provided with Windows. Therefore, once Wine is finished, you will not need to have windows installed to use Wine.

Wine has already made enough progress that it may be possible to run your target applications without Windows installed. If you want to try it, follow these steps:

1. Create empty `C:\windows`, `C:\windows\system`, `C:\windows\Start Menu`, and `C:\windows\Start Menu\Programs` directories. Do not point Wine to a `Windows` directory full of old installations and a messy registry. (Wine creates a

special registry in your `home` directory, in `$HOME/.wine/*.reg`. Perhaps you have to remove these files).

2. Point *[Drive C]* in `wine.conf` or `.winerc` to where you want `C:` to be. Refer to the Wine man page for more information. Remember to use **`filesystem=win95`**!
3. Use `tools/wineinstall` to compile Wine and install the default registry. Or if you prefer to do it yourself, compile `programs/regapi`, and run: **programs/regapi/regapi setValue < winedefault.reg**
4. Run and/or install your applications.

Because Wine is not yet complete, some programs will work better with native Windows DLL's than with Wine's replacements. Wine has been designed to make this possible. Here are some tips by Juergen Schmied (and others) on how to proceed. This assumes that your `C:\windows` directory in the configuration file does not point to a native Windows installation but is in a separate Unix file system. (For instance, "C:\windows" is really subdirectory "windows" located in "/home/ego/wine/drives/c").

- Run the application with *`--debugmsg +module,+file`* to find out which files are needed. Copy the required DLL's one by one to the `C:\windows\system` directory. Do not copy KERNEL/KERNEL32, GDI/GDI32, or USER/USER32. These implement the core functionality of the Windows API, and the Wine internal versions must be used.
- Edit the "[DllOverrides]" section of `wine.conf` or `.winerc` to specify "native" before "builtin" for the Windows DLL's you want to use. For more information about this, see the Wine manpage.
- Note that some network DLL's are not needed even though Wine is looking for them. The Windows `MPR.DLL` currently does not work; you must use the internal implementation.
- Copy SHELL/SHELL32 and COMDLG/COMDLG32 COMMCTRL/COMCTL32 only as pairs to your Wine directory (these DLL's are "clean" to use). Make sure you have these specified in the "[DllPairs]" section of `wine.conf` or .winerc.

- Be consistent: Use only DLL's from the same Windows version together.
- Put `regedit.exe` in the `C:\windows` directory (office95 imports a `*.reg` file when it runs with a empty registry, don't know about office97).
- Also add `winhelp.exe` and `winhlp32.exe` if you want to be able to browse through your programs' help function.

2.3. Dealing With FAT/VFAT Partitions

written by Steven Elliott (elliotsl@mindspring.com)

(Extracted from `wine/documentation/linux-fat-permissions`)

This document describes how FAT and VFAT file system permissions work in Linux with a focus on configuring them for Wine.

2.3.1. Introduction

Linux is able to access DOS and Windows file systems using either the FAT (older 8.3 DOS filesystems) or VFAT (newer Windows 95 or later long filename filesystems) modules. Mounted FAT or VFAT filesystems provide the primary means for which existing applications and their data are accessed through Wine for dual boot (Linux + Windows) systems.

Wine maps mounted FAT filesystems, such as `/c`, to driver letters, such as "c:", as indicated by the `wine.conf` file. The following excerpt from a `wine.conf` file does this:

```
[Drive C]
Path=/c
Type=hd
```

Although VFAT filesystems are preferable to FAT filesystems for their long filename support the term "FAT" will be used throughout the remainder of this document to refer

to FAT filesystems and their derivatives. Also, “/c” will be used as the FAT mount point in examples throughout this document.

Most modern Linux distributions either detect or allow existing FAT file systems to be configured so that can be mounted, in a location such as `/c`, either persistently (on bootup) or on an as needed basis. In either case, by default, the permissions will probably be configured so that they look something like:

```
~>cd /c
/c>ls -l
-rwxr-xr-x   1 root     root           91 Oct 10 17:58 autoexec.bat
-rwxr-xr-x   1 root     root          245 Oct 10 17:58 config.sys
drwxr-xr-x  41 root     root        16384 Dec 30  1998 windows
```

where all the files are owned by "root", are in the "root" group and are only writable by "root" (`755` permissions). This is restrictive in that it requires that Wine be run as root in order for applications to be able to write to any part of the filesystem.

There three major approaches to overcoming the restrictive permissions mentioned in the previous paragraph:

1. Run Wine as root
2. Mount the FAT filesystem with less restrictive permissions
3. Shadow the FAT filesystem by completely or partially copying it

Each approach will be discussed in the following sections.

2.3.2. Running Wine as root

Running Wine as root is the easiest and most thorough way of giving applications that Wine runs unrestricted access to FAT files systems. Running wine as root also allows applications to do things unrelated to FAT filesystems, such as listening to ports that are less than 1024. Running Wine as root is dangerous since there is no limit to what the application can do to the system.

2.3.3. Mounting FAT filesystems

The FAT filesystem can be mounted with permissions less restrictive than the default. This can be done by either changing the user that mounts the FAT filesystem or by explicitly changing the permissions that the FAT filesystem is mounted with. The permissions are inherited from the process that mounts the FAT filesystem. Since the process that mounts the FAT filesystem is usually a startup script running as root the FAT filesystem inherits root's permissions. This results in the files on the FAT filesystem having permissions similar to files created by root. For example:

```
~>whoami
root
~>touch root_file
~>ls -l root_file
-rw-r--r--   1 root     root            0 Dec 10 00:20 root_file
```

which matches the owner, group and permissions of files seen on the FAT filesystem except for the missing 'x's. The permissions on the FAT filesystem can be changed by changing root's umask (unset permissions bits). For example:

```
~>umount /c
~>umask
022
~>umask 073
~>mount /c
~>cd /c
/c>ls -l
-rwx---r--   1 root     root           91 Oct 10 17:58 autoexec.bat
-rwx---r--   1 root     root          245 Oct 10 17:58 config.sys
drwx---r--  41 root     root        16384 Dec 30  1998 windows
```

Mounting the FAT filesystem with a umask of `000` gives all users complete control over it. Explicitly specifying the permissions of the FAT filesystem when it is mounted provides additional control. There are three mount options that are relevant to FAT

permissions: `uid`, `gid` and `umask`. They can each be specified when the filesystem is manually mounted. For example:

```
~>umount /c
~>mount -o uid=500 -o gid=500 -o umask=002 /c
~>cd /c
/c>ls -l
-rwxrwxr-x   1 sle      sle            91 Oct 10 17:58 autoexec.bat
-rwxrwxr-x   1 sle      sle           245 Oct 10 17:58 config.sys
drwxrwxr-x  41 sle      sle         16384 Dec 30  1998 windows
```

which gives "sle" complete control over `/c`. The options listed above can be made permanent by adding them to the `/etc/fstab` file:

```
~>grep /c /etc/fstab
/dev/hda1  /c  vfat  uid=500,gid=500,umask=002,exec,dev,suid,rw 1 1
```

Note that the umask of `002` is common in the user private group file permission scheme. On FAT file systems this umask assures that all files are fully accessible by all users in the specified group (`gid`).

2.3.4. Shadowing FAT filesystems

Shadowing provides a finer granularity of control. Parts of the original FAT filesystem can be copied so that the application can safely work with those copied parts while the application continue to directly read the remaining parts. This is done with symbolic links. For example, consider a system where an application named AnApp must be able to read and write to the `c:\windows` and `c:\AnApp` directories as well as have read access to the entire FAT filesystem. On this system the FAT filesystem has default permissions which should not be changed for security reasons or can not be changed due to lack of root access. On this system a shadow directory might be set up in the following manner:

```
~>cd /
/>mkdir c_shadow
/>cd c_shadow
/c_shadow>ln -s /c_/* .
/c_shadow>rm windows AnApp
/c_shadow>cp -R /c_/{windows,AnApp} .
/c_shadow>chmod -R 777 windows AnApp
/c_shadow>perl -p -i -
e 's|/c$|/c_shadow|g' /usr/local/etc/wine.conf
```

The above gives everyone complete read and write access to the `windows` and `AnApp` directories while only root has write access to all other directories.

2.4. SCSI Support

written by Bruce Milner; Additions by Andreas Mohr

(Extracted from `wine/documentation/aspi`)

This file describes setting up the Windows ASPI interface.

Warning/Warning/Warning!!!!!!

THIS MAY TRASH YOUR SYSTEM IF USED INCOR-
RECTLY
THIS MAY TRASH YOUR SYS-
TEM IF USED CORRECTLY

Now that I have said that. ASPI is a direct link to SCSI devices from windows programs. ASPI just forwards the SCSI commands that programs send to it to the SCSI

bus.

If you use the wrong scsi device in your setup file, you can send completely bogus commands to the wrong device - An example would be formatting your hard drives (assuming the device gave you permission - if you're running as root, all bets are off).

So please make sure that **all** SCSI devices not needed by the program have their permissions set as restricted as possible !

Cookbook for setting up scanner: (At least how mine is to work)

2.4.1. Windows requirements

1. The scanner software needs to use the "Adaptec" compatible drivers (ASPI). At least with Mustek, they allow you the choice of using the builtin card or the "Adaptec (AHA)" compatible drivers. This will not work any other way. Software that accesses the scanner via a DOS ASPI driver (e.g. ASPI2DOS) is supported, too. [AM]
2. You probably need a real windows install of the software to set the LUN's/SCSI id's up correctly. I'm not exactly sure.

2.4.2. LINUX requirements:

1. Your scsi card must be supported under linux. This will not work with an unknown scsi card. Even for cheap'n crappy "scanner only" controllers some special Linux drivers exist on the net.
2. Compile generic scsi drivers into your kernel.
3. Linux by default uses smaller scsi buffers than Windows. There is a kernel build define `SG_BIG_BUFF` (in `sg.h`) that is by default set too low. The SANE project recommends `130560` and this seems to work just fine. This does require a kernel rebuild.

4. Make the devices for the scanner (generic scsi devices) - look at the scsi programming how-to for device numbering.
5. I would recommend making the scanner device writable by a group. I made a group called `scanner` and added myself to it. Running as root increases your risk of sending bad scsi commands to the wrong device. With a regular user, you are better protected.
6. Add a scsi device entry for your particular scanner to wine.conf. The format is `[scsi cCtTdD]` where `C=controller`, `T=target`, `D=LUN`

   For example, I set mine up as controller `0`, Target `6`, LUN `0`.

   ```
   [scsi c0t6d0]
   Device=/dev/sgi
   ```

   Yours will vary with your particular SCSI setup.

2.4.3. General Information

The mustek scanner I have was shipped with a package "ipplus". This program uses the TWAIN driver specification to access scanners.

(TWAIN MANAGER)

```
ipplus.exe <---> (TWAIN INTERFACE) <---
> (TWAIN DATA SOURCE . ASPI) -> WINASPI
```

2.4.4. NOTES/BUGS

The biggest is that it only works under linux at the moment.

The ASPI code has only been tested with:

- a Mustek 800SP with a Buslogic controller under Linux [BM]
- a Siemens Nixdorf 9036 with Adaptec AVA-1505 under Linux accessed via DOSASPI. Note that I had color problems, though (barely readable result) [AM]
- a Fujitsu M2513A MO drive (640MB) using generic scsi drivers. Formatting and ejecting worked perfectly. Thanks to Uwe Bonnes for access to the hardware ! [AM]

I make no warranty to the aspi code. It makes my scanner work. Your devices may explode. I have no way of determining this. I take zero responsibility!

Chapter 3. Configuring Wine

Setting up config files, etc.

3.1. General Configuration

Copyright 1999 Adam Sacarny (magicbox@bestweb.net)

(Extracted from `wine/documentation/config`)

3.1.1. The Wine Config File

The Wine config file stores various settings for Wine. These include:

- Drives and Information about them
- Directory Settings
- Port Settings
- The Wine look and feel
- Wine's DLL Usage

3.1.2. How Do I Make One?

This section will guide you through the process of making a config file. Take a look at the file `<dirs to wine>/wine.ini`. It is organized by section.

Section Name	Needed?	What it Does

Section Name	Needed?	What it Does
[Drive X]	yes	Sets up drives recognized by wine
[wine]	yes	Settings for wine directories
[DllDefaults]	recmd	Defaults for loading DLL's
[DllPairs]	recmd	Sanity checkers for DLL's
[DllOverrides]	recmd	Overides defaults for DLL loading
[options]	no	No one seems to know
[fonts]	yes	Font appearance and recognition
[serialports]	no	COM ports seen by wine
[parallelports]	no	LPT ports seen by wine
[spooler]	no	Print spooling
[ports]	no	Direct port access
[spy]	no	What to do with certain debug messages
[Registry]	no	Specifies locations of windows registry files
[tweak.layout]	recmd	Appearance of wine
[programs]	no	Programs to be run automatically
[Console]	no	Console settings

3.1.2.1. The [Drive X] Section

It should be pretty self explanatory, but here is an in-depth tutorial about them. There are up to 6 lines for each drive in Wine.

```
[Drive X]
```

The above line begins the section for a drive whose letter is X.

```
Path=/dir/to/path
```

This path is where the drive will begin. When Wine is browsing in drive X, it will see the files that are in the directory `/dir/to/path`. Don't forget to leave off the trailing slash!

```
Type=floppy|hd|cdrom|network <---
the |'s mean Type=<one of the options>
```

Sets up the type of drive Wine will see it as. Type must equal one of the four `floppy`, `hd`, `cdrom`, or `network`. They are self-explanatory.

```
Label=blah
```

Defines the drive label. Generally only needed for programs that look for a special CD-ROM. Info on finding the lable is in `<dirs to wine>/documentation/cdrom-labels`. The label may be up to 11 characters.

```
Serial=deadbeef
```

Tells Wine the serial number of the drive. A few programs with intense protection for pirating might need this, but otherwise don't use it. Up to 8 characters and hexadecimal.

```
Filesystem=msdos|win95|unix
```

Sets up the way Wine looks at files on the drive.

`msdos`

Case insensitive filesystem. Alike to DOS and Windows 3.x. `8.3` is the maximum length of files (eightdot.123) - longer ones will be truncated. (NOTE: this is a very bad choice if you plan on running apps that use long filenames. win95 should work fine with apps that were designed to run under the msdos system. In other words, you might not want to use this.)

`win95`

Case insensitive. Alike to Windows 9x/NT 4. This is the long filename filesystem you are probably used to working with. The filesystem of choice for most applications to be run under wine. PROBABLY THE ONE YOU WANT!

`unix`

Case sensitive. This filesystem has almost no use (Windows apps expect case insensitive filenames). Try it if you dare, but win95 is a much better choice.

`Device=/dev/xx`

Use this ONLY for floppy and cdrom devices. Using it on Extended2 partitions can have dire results (when a windows app tries to do a lowlevel write, they do it in a FAT way -- FAT does not mix with Extended2).

Note: This setting is not really important; almost all apps will have no problem if it remains unspecified. For CD-ROMs you might want to add it to get automatic label detection, though. If you are unsure about specifying device names, just leave out this setting for your drives.

Here is a setup for Drive X, a generic hard drive:

```
[Drive X]
Path=/dos-a
Type=hd
Label=Hard Drive
Filesystem=win95
```

```
This is a setup for Drive X, a generic CD-ROM drive:
[Drive X]
Path=/dos-d
Type=cdrom
Label=Total Annihilation
Filesystem=win95
Device=/dev/hdc
And here is a setup for Drive X, a generic floppy drive:
[Drive X]
Type=floppy
Path=/mnt/floppy
Label=Floppy Drive
Serial=87654321
Filesystem=win95
Device=/dev/fd0
```

3.1.2.2. The [wine] Section

The [wine] section of the configuration file contains information wine uses for directories. When specifying the directories for the settings, make them as they would appear in wine. If your drive *C* has a path of `/dos`, and your `windows` directory is located in `/dos/windows`, then use:

```
Windows=c:\windows
```

This sets up the `windows` directory. Make one if you don't already have one. NO TRAILING SLASH (NOT `C:\windows\`)!

```
System=c:\windows\system
```

This sets up where the windows system files are. Should reside in the directory used for the `Windows` setting. If you don't have `windows` then this is where the system files will go. Again, NO TRAILING SLASH!

```
Temp=c:\temp
```

This should be the directory you want your temp files stored in. YOU MUST HAVE WRITE ACCESS TO IT.

```
Path=c:\windows;c:\windows\system;c:\blanco
```

Behaves like the PATH setting on UNIX boxes. When wine is run like **`wine sol.exe`**, if `sol.exe` resides in a directory specified in the `Path` setting, wine will run it (Of course, if `sol.exe` resides in the current directory, wine will run that one). Make sure it always has your `windows` directory and system directory (For this setup, it must have `c:\windows;c:\windows\system`).

```
SymbolTableFile=wine.sym
```

Sets up the symbol table file for the wine debugger. You probably don't need to fiddle with this. May be useful if your wine is stripped.

```
printer=off|on
```

Tells wine whether to allow printer drivers and printing to work. Using these things are pretty alpha, so you might want to watch out. Some people might find it useful, however. If you're not planning on working on printing, don't even add this to your `wine.ini` (It probably isn't already in it). Check out the [spooler] and [parallelports] sections too.

3.1.2.3. Introduction To DLL Sections

There are a few things you will need to know before configuring the DLL sections in your wine configuration file.

3.1.2.3.1. Windows DLL Pairs

Most windows DLL's have a win16 (Windows 3.x) and win32 (Windows 9x/NT) form. The combination of the win16 and win32 DLL versions are called the "DLL pair". This is a list of the most common pairs:

3.1.2.3.2. Different Forms Of DLL's

There are a few different forms of DLL's wine can load:

native

The DLL's that are included with windows. Many windows DLL's can be loaded in their native form. Many times these native versions work better than their non-Microsoft equivalent -- other times they don't.

elfdll

ELF encapsulated windows DLL's. This is currently experimental (Not working yet).

so

Native ELF libraries. Will not work yet.

builtin

The most common form of DLL loading. This is what you will use if the DLL is error-prone in native form (KERNEL for example), you don't have the native DLL, or you just want to be Microsoft-free.

3.1.2.4. The [DllDefaults] Section

These settings provide wine's default handling of DLL loading.

```
EXTRA_LD_LIBRARY_PATH=/dirs
```

The directory specified here is appended to the normal search path for certain forms of DLL's (elfdll and .so).

```
DefaultLoadOrder = native, elfdll, so, builtin
```

This setting is a comma-delimited list of which order to attempt loading DLL's. If the first option fails, it will try the second, and so on. The order specified above is probably the best in most conditions.

3.1.2.5. The [DllPairs] Section

This section is optional, but strongly recommended. If you try to use native SHELL32, but builtin SHELL, you could have some big problems (native and builtin/so/elfdll do certain things in different ways). Using different forms of a pair is a *very*, **very** bad idea. By specifying DLL pairs here, wine will print out a message if you use different forms of a pair. You shouldn't need to change anything in this section, the following should work fine in all cases:

```
[DllPairs]
kernel = kernel32
gdi = gdi32
user = user32
commdlg = comdlg32
commctrl= comctl32
ver = version
shell = shell32
lzexpand= lz32
winsock = wsock32
```

3.1.2.6. The [DllOverrides] Section

The format for this section is the same for each line:

```
<DLL>{,<DLL>,<DLL>...} = <FORM>{,<FORM>,<FORM>...}
```

For example, to load builtin KERNEL pair (Case doesn't matter here):

```
kernel,kernel32 = builtin
```

To load the native COMMDLG pair, but if that doesn't work try builtin:

```
commdlg,comdlg32 = native,builtin
```

To load the native COMCTL32:

```
comctl32 = native
```

Here is a good generic setup (As it is defined in wine.ini that was included with your wine package):

```
[DllOverrides]
kernel32, gdi32, user32 = builtin
kernel, gdi, user = builtin
toolhelp = builtin
comdlg32, commdlg = elfdll, builtin, native
version, ver = elfdll, builtin, native
shell32, shell = builtin, native
lz32, lzexpand = builtin, native
commctrl, comctl32 = builtin, native
wsock32, winsock = builtin
advapi32, crtdll, ntdll = builtin, native
mpr, winspool = builtin, native
ddraw, dinput, dsound = builtin, native
winmm, w32skrnl, msvfw32= builtin
wnaspi32, wow32 = builtin
system, display, wprocs = builtin
wineps = builtin
```

Note: You see that elfdll or so is the first option for a few of these dll's. This will fail for you, but you won't notice it as wine will just use the second or third option.

3.1.2.7. The [options] Section

No one seems to know what this section is...

```
AllocSystemColors=100
```

System colors to allocate? Just leave it at 100.

3.1.2.8. The [fonts] Section

This section sets up wine's font handling.

```
Resolution = 96
```

Since the way X handles fonts is different from the way Windows does, wine uses a special mechanism to deal with them. It must scale them using the number defined in the "Resolution" setting. 60-120 are reasonable values, 96 is a nice in the middle one. If you have the real windows fonts available (`<dirs to wine>/documentation/ttfserver` and `fonts`), this parameter will not be as important. Of course, it's always good to get your X fonts working acceptably in wine.

```
Default = -adobe-times-
```

The default font wine uses. Fool around with it if you'd like.

OPTIONAL:

The `Alias` setting allows you to map an X font to a font used in wine. This is good for apps that need a special font you don't have, but a good replacement exists. The syntax

is like so:

```
AliasX = [Fake windows name],[Real X name]<,optional "mask-
ing" section>
```

Pretty straightforward. Replace "AliasX" with "Alias0", then "Alias1" and so on. The fake windows name is the name that the font will be under a windows app in wine. The real X name is the font name as seen by X (Run "xfontsel"). The optional "masking" section allows you to utilize the fake windows name you define. If it is not used, then wine will just try to extract the fake windows name itself and not use the value you enter.

Here is an example of an alias without masking. The font will show up in windows apps as "Google". When defining an alias in a config file, forget about my comment text (The "<-- blah" stuff)

```
Alias0 = Foo,--google-      <--
Note the no spaces after the " = ". Important!
```

Here is an example with masking enabled. The font will show up as "Foo" in windows apps.

```
Alias1 = Foo,--google-,subst
```

For more info check out `<dirs to wine>/documentation/fonts`

3.1.2.9. The [serialports], [parallelports], [spooler], and [ports] Sections

Even though it sounds like a lot of sections, these are all closely related. They all are for communications and parallel ports.

The [serialports] section tells wine what serial ports it is allowed to use.

```
ComX=/dev/cuaY
```

Replace `X` with the number of the COM port in Windows (1-8) and `Y` with the number of it in `X` (Usually the number of the port in Windows minus 1). `ComX` can actually equal any device (*/dev/modem* is acceptable). It is not always necessary to define any COM ports (An optional setting). Here is an example:

```
Com1=/dev/cua0
```

Use as many of these as you like in the section to define all of the COM ports you need.

The [parallelports] section sets up any parallel ports that will be allowed access under wine.

```
LptX=/dev/lpY
```

Seem farmiliar? Syntax is just like the COM port setting. Replace `X` with a value from 1-4 as it is in Windows and `Y` with a value from 0-3 (`Y` is usually the value in windows minus 1, just like for COM ports). You don't always need to define a parallel port (AKA, it's optional). As with the other section, LptX can equal any device (Maybe */dev/printer*). Here is an example:

```
Lpt1=/dev/lp0
```

The [spooler] section will inform wine where to spool print jobs. Use this if you want to try printing. Wine docs claim that spooling is "rather primitive" at this time, so it won't work perfectly. IT IS OPTIONAL. The only setting you use in this section works to map a port (LPT1, for example) to a file or a command. Here is an example, mapping LPT1 to the file `out.ps`:

```
LPT1:=out.ps
```

The following command maps printing jobs to LPT1 to the command **lpr**. Notice the |:

```
LPT1:=|lpr
```

The [ports] section is usually useful only for people who need direct port access for programs requiring dongles or scanners. IF YOU DON'T NEED IT, DON'T USE IT!

```
read=0x779,0x379,0x280-0x2a0
```

Gives direct read access to those IO's.

```
write=0x779,0x379,0x280-0x2a0
```

Gives direct write access to those IO's. It probably a good idea to keep the values of the `read` and `write` settings the same. This stuff will only work when you're root.

3.1.2.10. The [spy], [Registry], [tweak.layout], and [programs] Sections

[spy] is used to Include or exclude debug messages, and to output them to a file. The latter is rarely used. THESE ARE ALL OPTIONAL AND YOU PROBABLY DON'T NEED TO ADD OR REMOVE ANYTHING IN THIS SECTION TO YOUR CONFIG.

```
File=/blanco
```

Sets the logfile for wine. Set to CON to log to standard out. THIS IS RARELY USED.

```
Exclude=WM_SIZE;WM_TIMER;
```

Excludes debug messages about `WM_SIZE` and `WM_TIMER` in the logfile.

```
Include=WM_SIZE;WM_TIMER;
```

Includes debug messages about `WM_SIZE` and `WM_TIMER` in the logfile.

[Registry] can be used to tell wine where your old windows registry files exist. This section is completely optional and useless to people using wine without an existing windows installation.

```
UserFileName=/dirs/to/user.reg
```

The location of your old `user.reg` file.

```
LocalMachineFileName=/dirs/to/system.reg
```

The location of your old `system.reg` file.

[tweak.layout] is devoted to wine's look. There is only one setting for it.

```
WineLook=win31|win95|win98
```

Will change the look of wine from Windows 3.1 to Windows 95. The `win98` setting behaves just like `win95` most of the time.

[programs] can be used to say what programs run under special conditions.

```
Default=/program/to/execute.exe
```

Sets the program to be run if wine is started without specifying a program.

```
Startup=/program/to/execute.exe
```

Sets the program to automatically be run at startup every time.

3.1.3. Where Do I Put It?

The wine config file can go in two places.

`/usr/local/etc/wine.conf`

A systemwide config file, used for anyone who doesn't have their own.

`$HOME/.winerc`

Your own config file, that only is used for your user.

So copy your version of the `wine.conf` file to `/usr/local/etc/wine.conf` or `$HOME/.winerc` for wine to recognize it.

3.1.4. What If It Doesn't Work?

There is always a chance that things will go wrong. If the unthinkable happens, try the newsgroup, comp.emulators.ms-windows.wine Make sure that you have looked over this document thoroughly, and have also read:

- `README`
- `documentation/bugreports`
- `http://www.westfalen.de/witch/wine-HOWTO.txt` (Optional but recommended)

If indeed it looks like you've done your research, be prepared for helpful suggestions. If you haven't, brace yourself for heaving flaming.

3.2. Win95/98 Look

by ???

(Extracted from `wine/documentation/win95look`)

Win95/Win98 interface code is being introduced.

Instead of compiling Wine for Win3.1 vs. Win95 using `#define` switches, the code now looks in a special [Tweak.Layout] section of `wine.conf` for a `WineLook=Win95` or `WineLook=Win98` entry.

A few new sections and a number of entries have been added to the `wine.conf file` -- these are for debugging the Win95 tweaks only and may be removed in a future release! These entries/sections are:

```
[Tweak.Fonts]
System.Height=<point size>    # Sets the height of the sys-
tem typeface
System.Bold=[true|false]      # Whether the sys-
tem font should be boldfaced
System.Italic=[true|false]    # Whether the sys-
tem font should be italicized
System.Underline=[true|false] # Whether the sys-
tem font should be underlined
System.StrikeOut=[true|false] # Whether the sys-
tem font should be struck out
OEMFixed.xxx                  # Same parame-
ters for the OEM fixed typeface
AnsiFixed.xxx                 # Same parame-
ters for the Ansi fixed typeface
AnsiVar.xxx                   # Same parame-
ters for the Ansi variable typeface
SystemFixed.xxx               # Same parameters for the Sys-
tem fixed typeface

[Tweak.Layout]
WineLook=[Win31|Win95|Win98]  # Changes Wine's look and feel
```

3.3. Configuring the x11drv Driver

written by Ove Kåven

(Extracted from `wine/documentation/x11drv`)

Most Wine users run Wine under the windowing system known as X11. During most of Wine's history, this was the only display driver available, but in recent years, parts of Wine has been reorganized to allow for other display drivers (although the only alternative currently available is Patrik Stridvall's ncurses-based ttydrv, which he claims works for displaying calc.exe). The display driver is chosen with the `GraphicsDriver` option in the [wine] section of `wine.conf` or `.winerc`, but I will only cover the x11drv driver in this article.

3.3.1. x11drv modes of operation

The x11drv driver consists of two conceptually distinct pieces, the graphics driver (GDI part), and the windowing driver (USER part). Both of these are linked into the `libx11drv.so` module, though (which you load with the `GraphicsDriver` option). In Wine, running on X11, the graphics driver must draw on drawables (window interiors) provided by the windowing driver. This differs a bit from the Windows model, where the windowing system creates and configures device contexts controlled by the graphics driver, and applications are allowed to hook into this relationship anywhere they like. Thus, to provide any reasonable tradeoff between compatibility and usability, the x11drv has three different modes of operation.

Unmanaged/Normal

> The default. Window-manager-independent (any running window manager is ignored completely). Window decorations (title bars, borders, etc) are drawn by Wine to look and feel like the real Windows. This is compatible with applications that depend on being able to compute the exact sizes of any such decorations, or that want to draw their own.

Managed

> Specified by using the *`--managed`* command-line option or the `Managed` `wine.conf` option (see below). Ordinary top-level frame windows with thick borders, title bars, and system menus will be managed by your window manager.

This lets these applications integrate better with the rest of your desktop, but may not always work perfectly. (A rewrite of this mode of operation, to make it more robust and less patchy, is highly desirable, though, and is planned to be done before the Wine 1.0 release.)

Desktop-in-a-Box

Specified by using the *--desktop* command-line option (with a geometry, e.g. *--desktop 800x600* for a such-sized desktop, or even *--desktop 800x600+0+0* to automatically position the desktop at the upper-left corner of the display). This is the mode most compatible with the Windows model. All application windows will just be Wine-drawn windows inside the Wine-provided desktop window (which will itself be managed by your window manager), and Windows applications can roam freely within this virtual workspace and think they own it all, without disturbing your other X apps.

3.3.2. The [x11drv] section

AllocSystemColors

Applies only if you have a palette-based display, i.e. if your X server is set to a depth of 8bpp, and if you haven't requested a private color map. It specifies the maximum number of shared colormap cells (palette entries) Wine should occupy. The higher this value, the less colors will be available to other applications.

PrivateColorMap

Applies only if you have a palette-based display, i.e. if your X server is set to a depth of 8bpp. It specifies that you don't want to use the shared color map, but a private color map, where all 256 colors are available. The disadvantage is that Wine's private color map is only seen while the mouse pointer is inside a Wine window, so psychedelic flashing and funky colors will become routine if you use the mouse a lot.

PerfectGraphics

This option only determines whether fast X11 routines or exact Wine routines will be used for certain ROP codes in blit operations. Most users won't notice any difference.

ScreenDepth

Applies only to multi-depth displays. It specifies which of the available depths Wine should use (and tell Windows apps about).

Display

This specifies which X11 display to use, and if specified, will override both the DISPLAY environment variable and the `--display` command-line option.

Managed

Wine can let frame windows be managed by your window manager. This option specifies whether you want that by default.

UseDGA

This specifies whether you want DirectDraw to use XFree86's *Direct Graphics Architecture* (DGA), which is able to take over the entire display and run the game full-screen at maximum speed. (With DGA1 (XFree86 3.x), you still have to configure the X server to the game's requested bpp first, but with DGA2 (XFree86 4.x), runtime depth-switching may be possible, depending on your driver's capabilities.) But be aware that if Wine crashes while in DGA mode, it may not be possible to regain control over your computer without rebooting. DGA normally requires either root privileges or read/write access to `/dev/mem`.

UseXShm

If you don't want DirectX to use DGA, you can at least use X Shared Memory extensions (XShm). It is much slower than DGA, since the app doesn't have direct access to the physical frame buffer, but using shared memory to draw the frame is at least faster than sending the data through the standard X11 socket, even though Wine's XShm support is still known to crash sometimes.

DXGrab

If you don't use DGA, you may want an alternative means to convince the mouse cursor to stay within the game window. This option does that. Of course, as with DGA, if Wine crashes, you're in trouble (although not as badly as in the DGA case, since you can still use the keyboard to get out of X).

DesktopDoubleBuffered

Applies only if you use the *--desktop* command-line option to run in a desktop window. Specifies whether to create the desktop window with a double-buffered visual, something most OpenGL games need to run correctly.

3.4. The Registry

written by Ove Kåven

(Extracted from `wine/documentation/registry`)

After Win3.x, the registry became a fundamental part of Windows. It is the place where both Windows itself, and all Win95/98/NT/2000/whatever-compliant applications, store configuration and state data. While most sane system administrators (and Wine developers) curse badly at the twisted nature of the Windows registry, it is still necessary for Wine to support it somehow.

3.4.1. Registry structure

The Windows registry is an elaborate tree structure, and not even most Windows programmers are fully aware of how the registry is laid out, with its different "hives" and numerous links between them; a full coverage is out of the scope of this document. But here are the basic registry keys you might need to know about for now.

HKEY_LOCAL_MACHINE

This fundamental root key (in win9x, stored in the hidden file `system.dat`) contains everything pertaining to the current Windows installation.

HKEY_USERS

This fundamental root key (in win9x, stored in the hidden file `user.dat`) contains configuration data for every user of the installation.

HKEY_CLASSES_ROOT

This is a link to HKEY_LOCAL_MACHINE\Software\Classes. It contains data describing things like file associations, OLE document handlers, and COM classes.

HKEY_CURRENT_USER

This is a link to HKEY_USERS\your_username, i.e., your personal configuration.

3.4.2. Using a Windows registry

If you point Wine at an existing MS Windows installation (by setting the appropriate directories in `wine.conf` or `.winerc`), then Wine is able to load registry data from it. However, Wine will not save anything to the real Windows registry, but rather to its own registry files (see below). Of course, if a particular registry value exists in both the Windows registry and in the Wine registry, then Wine will use the latter.

Occasionally, Wine may have trouble loading the Windows registry. Usually, this is because the registry is inconsistent or damaged in some way. If that becomes a problem, you may want to download the `regclean.exe` from the MS website and use it to clean up the registry. Alternatively, you can always use `regedit.exe` to export the registry data you want into a text file, and then import it in Wine.

3.4.3. Wine registry data filesr

In the user's home directory, there is a subdirectory named `.wine`, where Wine will try to save its registry by default. It saves into four files, which are:

`system.reg`

This file contains HKEY_LOCAL_MACHINE.

`user.reg`

This file contains HKEY_CURRENT_USER.

`userdef.reg`

This file contains HKEY_USERS\.Default (i.e. the default user settings).

`wine.userreg`

Wine saves HKEY_USERS to this file (both current and default user), but does not load from it, unless `userdef.reg` is missing.

All of these files are human-readable text files, so unlike Windows, you can actually use an ordinary text editor on them if you must.

In addition to these files, Wine can also optionally load from global registry files residing in the same directory as the global `wine.conf` (i.e. `/usr/local/etc` if you compiled from source). These are:

`wine.systemreg`

Contains HKEY_LOCAL_MACHINE.

`wine.userreg`

Contains HKEY_USERS.

3.4.4. System administration

With the above file structure, it is possible for a system administrator to configure the system so that a system Wine installation (and applications) can be shared by all the users, and still let the users all have their own personalized configuration. An administrator can, after having installed Wine and any Windows application software he wants the users to have access to, copy the resulting `system.reg` and `wine.userreg` over to the global registry files (which we assume will reside in `/usr/local/etc` here), with:

```
cd ~/.wine
cp system.reg /usr/local/etc/wine.systemreg
cp wine.userreg /usr/local/etc/wine.userreg
```

and perhaps even symlink these back to the administrator's account, to make it easier to install apps system-wide later:

```
ln -sf /usr/local/etc/wine.systemreg system.reg
ln -sf /usr/local/etc/wine.userreg wine.userreg
```

Note that the `tools/wineinstall` script already does all of this for you, if you install Wine as root. If you then install Windows applications while logged in as root, all your users will automatically be able to use them. While the application setup will be taken from the global registry, the users' personalized configurations will be saved in their own home directories.

But be careful with what you do with the administrator account - if you do copy or link the administrator's registry to the global registry, any user might be able to read the administrator's preferences, which might not be good if sensitive information (passwords, personal information, etc) is stored there. Only use the administrator account to install software, not for daily work; use an ordinary user account for that.

3.4.5. The default registry

A Windows registry contains many keys by default, and some of them are necessary for even installers to operate correctly. The keys that the Wine developers have found necessary to install applications are distributed in a file called `winedefault.reg`. It is automatically installed for you if you use the `tools/wineinstall` script, but if you want to install it manually, you can do so by using the **regapi** tool. You can find more information about this in the `documentation/no-windows` document in the Wine distribution.

3.4.6. The [registry] section

With the above information fresh in mind, let's look at the `wine.conf`/`.winerc` options for handling the registry.

LoadGlobalRegistryFiles

Controls whether to try to load the global registry files, if they exist.

LoadHomeRegistryFiles

Controls whether to try to load the user's registry files (in the `.wine` subdirectory of the user's home directory).

LoadWindowsRegistryFiles

Controls whether Wine will attempt to load registry data from a real Windows registry in an existing MS Windows installation.

WritetoHomeRegistryFiles

Controls whether registry data will be written to the user's registry files. (Currently, there is no alternative, so if you turn this off, Wine cannot save the registry on disk at all; after you exit Wine, your changes will be lost.)

UseNewFormat

This option is obsolete. Wine now always use the new format; support for the old format was removed a while ago.

PeriodicSave

If this option is set to a nonzero value, it specifies that you want the registry to be saved to disk at the given interval. If it is not set, the registry will only be saved to disk when the wineserver terminates.

SaveOnlyUpdatedKeys

Controls whether the entire registry is saved to the user's registry files, or only subkeys the user have actually changed. Considering that the user's registry will override any global registry files and Windows registry files, it usually makes sense to only save user-modified subkeys; that way, changes to the rest of the global or Windows registries will still affect the user.

3.5. Drive labels and serial numbers with wine

by Petr Tomasek <tomasek@etf.cuni.cz> Nov 14 1999

changes by Andreas Mohr <a.mohr@mailto.de> Jan 25 2000

(Extracted from `wine/documentation/cdrom-labels`)

Until now, your only possibility of specifying drive volume labels and serial numbers was to set them manually in the wine config file. By now, wine can read them directly from the device as well. This may be useful for many Win 9x games or for setup programs distributed on CD-ROMs that check for volume label.

3.5.1. What's Supported?

File System	Types	Comment
FAT systems	hd, floppy	reads labels and serial numbers
ISO9660	cdrom	reads labels only

3.5.2. How To Set Up?

Reading labels and serial numbers just works automagically if you specify a `Device=` line in the [Drive X] section in your `wine.conf`. Note that the device has to exist and must be accessible if you do this, though.

If you don't do that, then you should give fixed `Label=` or `Serial=` entries in `wine.conf`, as Wine returns these entries instead if no device is given. If they don't exist, then Wine will return default values (label `Drive X` and serial `12345678`).

If you want to give a `Device=` entry *only* for drive raw sector accesses, but not for reading the volume info from the device (i.e. you want a *fixed*, preconfigured label), you need to specify `ReadVolInfo=0` to tell Wine to skip the volume reading.

3.5.3. EXAMPLES

Here's a simple example of cdrom and floppy; labels will be read from the device on both cdrom and floppy; serial numbers on floppy only:

```
[Drive A]
Path=/mnt/floppy
Type=floppy
Device=/dev/fd0
Filesystem=msdos

[Drive R]
Path=/mnt/cdrom
Type=cdrom
```

```
Device=/dev/hda1
Filesystem=win95
```

Here's an example of overriding the CD-ROM label:

```
[Drive J]
Path=/mnt/cdrom
Type=cdrom
Label=X234GCDSE
; note that the device isn't re-
ally needed here as we have a fixed label
Device=/dev/cdrom
Filesystem=msdos
```

3.5.4. Todo / Open Issues

- The cdrom label can be read only if the data track of the disk resides in the first track and the cdrom is iso9660.
- Better checking for FAT superblock (it now checks only one byte).
- Support for labels/serial nums WRITING.
- Can the label be longer than 11 chars? (iso9660 has 32 chars).
- What about reading ext2 volume label? ....

3.6. Dll Overrides

by Ove Kaaven <ovek@arcticnet.no>

(Extracted from `wine/documentation/dll-overrides`)

The `wine.conf` directives [DllDefaults] and [DllOverrides] are the subject of some confusion. The overall purpose of most of these directives are clear enough, though - given a choice, should Wine use its own built-in DLLs, or should it use `.DLL` files found in an existing Windows installation? This document explains how this feature works.

3.6.1. DLL types

native

A "native" DLL is a `.DLL` file written for the real Microsoft Windows.

builtin

A "builtin" DLL is a Wine DLL. These can either be a part of `libwine.so`, or more recently, in a special `.so` file that Wine is able to load on demand.

elfdll

An "elfdll" is a Wine `.so` file with a special Windows-like file structure that is as close to Windows as possible, and that can also seamlessly link dynamically with "native" DLLs, by using special ELF loader and linker tricks. Bertho Stultiens did some work on this, but this feature has not yet been merged back into Wine (because of political reasons and lack of time), so this DLL type does not exist in the official Wine at this time. In the meantime, the "builtin" DLL type gained some of the features of elfdlls (such as dynamic loading), so it’s possible that "elfdll" functionality will be folded into "builtin" at some point.

so

A native Unix `.so` file, with calling convention conversion thunks generated on the fly as the library is loaded. This is mostly useful for libraries such as "glide" that has exactly the same API on both Windows and Unix.

3.6.2. The [DllDefaults] section

EXTRA_LD_LIBRARY_PATH

This specifies the location of the Wine's DLL `.so` files. Wine will search this path when trying to locate a DLL of the type `builtin` or `elfdll`. (This does not apply to `libwine.so`, since `libwine.so` is not a DLL in this sense.)

DefaultLoadOrder

This specifies in what order Wine should search for available DLL types, if the DLL in question was not found in the [DllOverrides] section.

3.6.3. The [DllPairs] section

At one time, there was a section called [DllPairs] in the default configuration file, but this has been obsoleted because the pairing information has now been embedded into Wine itself. (The purpose of this section was merely to be able to issue warnings if the user attempted to pair codependent 16-bit/32-bit DLLs of different types.) If you still have this in your `wine.conf` or `.winerc`, you may safely delete it.

3.6.4. The [DllOverrides] section

This section specifies how you want specific DLLs to be handled, in particular whether you want to use "native" DLLs or not, if you have some from a real Windows configuration. Because builtins do not mix seamlessly with native DLLs yet, certain DLL dependencies may be problematic, but workarounds exist in Wine for many popular DLL configurations. Also see WWN's [16]Status Page to figure out how well your favorite DLL is implemented in Wine.

It is of course also possible to override these settings by explictly using Wine's *`--dll`* command-line option (see the man page for details). Some hints for choosing your optimal configuration (listed by 16/32-bit DLL pair):

krnl386, kernel32

Native versions of these will never work, so don't try. Leave at `builtin`.

gdi, gdi32

Graphics Device Interface. No effort has been made at trying to run native GDI. Leave at `builtin`.

user, user32

Window management and standard controls. It was possible to use Win95's `native` versions at some point (if all other DLLs that depend on it, such as comctl32 and comdlg32, were also run `native`). However, this is no longer possible after the Address Space Separation, so leave at `builtin`.

ntdll

NT kernel API. Although badly documented, the `native` version of this will never work. Leave at `builtin`.

w32skrnl

Win32s (for Win3.x). The `native` version will probably never work. Leave at `builtin`.

wow32

Win16 support library for NT. The `native` version will probably never work. Leave at `builtin`.

system

Win16 kernel stuff. Will never work `native`. Leave at `builtin`.

display

Display driver. Definitely leave at `builtin`.

toolhelp

Tool helper routines. This is rarely a source of problems. Leave at `builtin`.

ver, version

Versioning. Seldom useful to mess with.

advapi32

Registry and security features. Trying the `native` version of this may or may not work.

commdlg, comdlg32

Common Dialogs, such as color picker, font dialog, print dialog, open/save dialog, etc. It is safe to try `native`.

commctrl, comctl32

Common Controls. This is toolbars, status bars, list controls, the works. It is safe to try `native`.

shell, shell32

Shell interface (desktop, filesystem, etc). Being one of the most undocumented pieces of Windows, you may have luck with the `native` version, should you need it.

winsock, wsock32

Windows Sockets. The `native` version will not work under Wine, so leave at `builtin`.

icmp

ICMP routines for wsock32. As with wsock32, leave at `builtin`.

mpr

The `native` version may not work due to thunking issues. Leave at `builtin`.

lzexpand, lz32

Lempel-Ziv decompression. Wine's `builtin` version ought to work fine.

winaspi, wnaspi32

Advanced SCSI Peripheral Interface. The `native` version will probably never work. Leave at `builtin`.

crtdll

C Runtime library. The `native` version will easily work better than Wine's on this one.

winspool.drv

Printer spooler. You are not likely to have more luck with the `native` version.

ddraw

DirectDraw/Direct3D. Since Wine does not implement the DirectX HAL, the `native` version will not work at this time.

dinput

DirectInput. Running this `native` may or may not work.

dsound

DirectSound. It may be possible to run this `native`, but don't count on it.

dplay/dplayx

DirectPlay. The `native` version ought to work best on this, if at all.

mmsystem, winmm

Multimedia system. The `native` version is not likely to work. Leave at `builtin`.

msacm, msacm32

Audio Compression Manager. The `builtin` version works best, if you set msacm.drv to the same.

msvideo, msvfw32

Video for Windows. It is safe (and recommended) to try `native`.

mcicda.drv

CD Audio MCI driver.

mciseq.drv

MIDI Sequencer MCI driver (`.MID` playback).

mciwave.drv

Wave audio MCI driver (`.WAV` playback).

mciavi.drv

AVI MCI driver (`.AVI` video playback). Best to use `native`.

mcianim.drv

Animation MCI driver.

msacm.drv

Audio Compression Manager. Set to same as msacm32.

midimap.drv

MIDI Mapper.

wprocs

This is a pseudo-DLL used by Wine for thunking purposes. A `native` version of this doesn't exist.

3.7. Keyboard

by Ove Kaaven <ovek@arcticnet.no>

(Extracted from `wine/documentation/keyboard`)

Wine now needs to know about your keyboard layout. This requirement comes from a need from many apps to have the correct scancodes available, since they read these directly, instead of just taking the characters returned by the X server. This means that Wine now needs to have a mapping from X keys to the scancodes these applications expect.

On startup, Wine will try to recognize the active X layout by seeing if it matches any of the defined tables. If it does, everything is alright. If not, you need to define it.

To do this, open the file `windows/x11drv/keyboard.c` and take a look at the existing tables. Make a backup copy of it, especially if you don't use CVS.

What you really would need to do, is find out which scancode each key needs to generate. Find it in the `main_key_scan` table, which looks like this:

```
static const int main_key_scan[MAIN_LEN] =
{
/* this is my (102-
key) keyboard layout, sorry if it doesn't quite match yours */
   0x29,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,
   0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,
   0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x2B,
   0x2C,0x2D,0x2E,0x2F,0x30,0x31,0x32,0x33,0x34,0x35,
   0x56 /* the 102nd key (actually to the right of l-shift) */
};
```

Next, assign each scancode the characters imprinted on the keycaps. This was done (sort of) for the US 101-key keyboard, which you can find near the top in `keyboard.c`. It also shows that if there is no 102nd key, you can skip that.

However, for most international 102-key keyboards, we have done it easy for you. The scancode layout for these already pretty much matches the physical layout in the

`main_key_scan`, so all you need to do is to go through all the keys that generate characters on your main keyboard (except spacebar), and stuff those into an appropriate table. The only exception is that the 102nd key, which is usually to the left of the first key of the last line (usually **Z**), must be placed on a separate line after the last line.

For example, my Norwegian keyboard looks like this

```
§  !  "  #  ¤  %  &  /  (  )  =  ?  `  Back-
|  1  2@ 3£ 4$ 5  6  7{ 8[ 9] 0} +  \´ space

Tab Q  W  E  R  T  Y  U  I  O  P  Å  ^
                                   ¨~
                                      Enter
Caps A  S  D  F  G  H  J  K  L  Ø  Æ  *
Lock                                  '

Sh- > Z  X  C  V  B  N  M  ;  :  _  Shift
ift <                     ,  .  -

Ctrl  Alt       Spacebar       AltGr  Ctrl
```

Note the 102nd key, which is the **<>** key, to the left of **Z**. The character to the right of the main character is the character generated by **AltGr**.

This keyboard is defined as follows:

```
static const char main_key_NO[MAIN_LEN][4] =
{
 "|§","1!","2\"@","3#£","4¤$","5%","6&","7/{","8([","9)]","0=}","+?","\\´",
 "qQ","wW","eE","rR","tT","yY","uU","iI","oO","pP","åÅ","¨^~",
 "aA","sS","dD","fF","gG","hH","jJ","kK","lL","øØ","æÆ","'*",
 "zZ","xX","cC","vV","bB","nN","mM",",;",".:","-_",
 "<>"
};
```

Except that " and \ needs to be quoted with a backslash, and that the 102nd key is on a separate line, it's pretty straightforward.

After you have written such a table, you need to add it to the `main_key_tab[]` layout index table. This will look like this:

```
static struct {
 WORD lang, ansi_codepage, oem_codepage;
 const char (*key)[MAIN_LEN][4];
} main_key_tab[]={
...
...
 {MAKE-
LANGID(LANG_NORWEGIAN,SUBLANG_DEFAULT),  1252, 865, &main_key_NO},
...
```

After you have added your table, recompile Wine and test that it works. If it fails to detect your table, try running

```
wine --debugmsg +key,+keyboard >& key.log
```

and look in the resulting `key.log` file to find the error messages it gives for your layout.

Note that the `LANG_*` and `SUBLANG_*` definitions are in `include/winnls.h`, which you might need to know to find out which numbers your language is assigned, and find it in the debugmsg output. The numbers will be `(SUBLANG * 0x400 + LANG)`, so, for example the combination `LANG_NORWEGIAN (0x14)` and `SUBLANG_DEFAULT (0x1)` will be (in hex) `14 + 1*400 = 414`, so since I'm Norwegian, I could look for `0414` in the debugmsg output to find out why my keyboard won't detect.

Once it works, submit it to the Wine project. If you use CVS, you will just have to do

```
cvs -z3 diff -u windows/x11drv/keyboard.c > layout.diff
```

from your main Wine directory, then submit `layout.diff` to <wine-patches@winehq.com> along with a brief note of what it is.

If you don't use CVS, you need to do

```
diff -u the_backup_file_you_made win-
dows/x11drv/keyboard.c > layout.diff
```

and submit it as explained above.

If you did it right, it will be included in the next Wine release, and all the troublesome applications (especially remote-control applications) and games that use scancodes will be happily using your keyboard layout, and you won't get those annoying fixme messages either.

Good luck.

Chapter 4. Finding and Reporting Bugs

4.1. How To Report A Bug

written by Gerard Patel (???)

(Extracted from `wine/documentation/bugreports`)

There are two ways for you to make a bug report. One uses a simple perl script, and is recommended if you don't want to spend a lot of time producing the report. It is designed for use by just about anyone, from the newest of newbies to advanced developers. You can also make a bug report the hard way -- advanced developers will probably prefer this.

4.1.1. The Easy Way

1. Your computer *must* have perl on it for this method to work. To find out if you have perl, run **which perl**. If it returns something like `/usr/bin/perl`, you're in business. Otherwise, skip on down to "The Hard Way". If you aren't sure, just keep on going. When you try to run the script, it will become *very* apparent if you don't have perl.
2. Change directory to `<dirs to wine>/tools`
3. Type in **./bug_report.pl** and follow the directions.
4. Post a message to the comp.emulators.ms-windows.wine newsgroup with the "Nice Formatted Report" attatched. If possible, upload the full debug output to a web/ftp server and provide the address in your message.

4.1.2. The Hard Way

Some simple advice on making your bug report more useful (and thus more likely to get answered and fixed):

1. Post as much information as possible.

   This means we need more information than a simple "MS Word crashes whenever I run it. Do you know why?" Include at least the following information:

   - Version of Wine you're using (run **wine -v**)
   - Operating system you're using, what distribution (if any), and what version
   - Compiler and version (run **gcc -v**)
   - Windows version, if installed
   - Program you're trying to run, its version number, and a URL for where the program can be obtained (if available)
   - Command line you used to start wine
   - Any other information you think may be relevant or helpful, such as X server version in case of X problems, libc version etc.

2. Re-run the program with the *--debugmsg +relay* option (i.e., **wine --debugmsg +relay sol.exe**).

   If Wine crashes while running your program, it is important that we have this information to have a chance at figuring out what is causing the crash. This can put out quite a lot (several MB) of information, though, so it's best to output it to a file. When the `Wine-dbg>` prompt appears, type **`quit`**.

   You might want to try *+relay,+snoop* instead of *+relay*, but please note that *+snoop* is pretty unstable and often will crash earlier than a simple *+relay*! If this is the case, then please use *only* *+relay*!! A bug report with a crash in *+snoop* code is useless in most cases!

   To get the trace output, use the following commands:

   all shells:

   ```
   $ echo quit | wine -debugmsg +relay [other_options] pro-
   gram_name >& filename.out;
   ```

```
$ tail -n 100 filename.out > report_file
```

(This will print wine's debug messages only to the file and then auto-quit. It's probably a good idea to use this command, since wine prints out so many debug msgs that they flood the terminal, eating CPU.)

tcsh and other csh-like shells:

```
    $ wine -debugmsg +relay [other_options] pro-
    gram_name |& tee filename.out;
$ tail -100 filename.out > report_file
```

bash and other sh-like shells:

```
    $ wine -debugmsg +relay [other_options] pro-
    gram_name 2>&1 | tee filename.out;
$ tail -100 filename.out > report_file
```

`report_file` will now contain the last hundred lines of the debugging output, including the register dump and backtrace, which are the most important pieces of information. Please do not delete this part, even if you don't understand what it means.

3. Post your report to the newsgroup comp.emulators.ms-windows.wine

   In your post, include all of the information from part 1), and insert the text from the output file in part 2). If you do this, your chances of receiving some sort of helpful response should be very good.

4.1.3. Questions and comments

If after reading this document there is something you couldn't figure out, or think could be explained better, or that should have been included, please post to comp.emulators.ms-windows.wine to let us know how this document can be improved.

Chapter 5. Dealing with Fonts

5.1. Fonts

written by ???

(Extracted from `wine/documentation/fonts`)

> **Note:** The **fnt2bdf** utility is included with Wine. It can be found in the `tools` directory. Links to the other tools mentioned in this document can be found on wine headquarters: http://www.winehq.com/tools.html

5.1.1. How To Convert Windows Fonts

If you have access to a Windows installation you should use **fnt2bdf** utility (found in the `tools` directory) to convert bitmap fonts (`VGASYS.FON`, `SSERIFE.FON`, and `SERIFE.FON`) into the format that the X Window System can recognize.

1. Extract bitmap fonts with **fnt2bdf**.
2. Convert `.bdf` files produced by Step 1 into `.pcf` files with **bdftopcf -o <target file> <original bdf-file>**.
3. Copy `.pcf` files to the font server directory which is usually `/usr/lib/X11/fonts/misc` (you will probably need superuser privileges). If you want to create a new font directory you will need to add it to the font path.
4. Run **mkfontdir** for the directory you copied fonts to. If you are already in X you should run **xset fp rehash** to make X server aware of the new fonts.
5. Edit the `wine.conf` file to remove aliases for the fonts you've just installed.

WINE can get by without these fonts but 'the look and feel' may be quite different. Also, some applications try to load their custom fonts on the fly (WinWord 6.0) and since WINE does not implement this yet it instead prints out something like;

```
STUB: AddFontResource( SOMEFILE.FON )
```

You can convert this file too. Note that `.FON` file may not hold any bitmap fonts and **fnt2bdf** will fail if this is the case. Also note that although the above message will not disappear WINE will work around the problem by using the font you extracted from the `SOMEFILE.FON`. **fnt2bdf** will only work for Windows 3.1 fonts. It will not work for TrueType fonts.

What to do with TrueType fonts? There are several commercial font tools that can convert them to the Type1 format but the quality of the resulting fonts is far from stellar. The other way to use them is to get a font server capable of rendering TrueType (Caldera has one, there also is the free **xfstt** in `Linux/X11/fonts` on sunsite and mirrors, if you're on FreeBSD you can use the port in `/usr/ports/x11-servers/Xfstt`. And there is **xfsft** which uses the freetype library, see `documentation/ttfserver`).

However, there is a possibility of the native TrueType support via FreeType renderer in the future (hint, hint :-)

5.1.2. How To Add Font Aliases To `wine.conf`

Many Windows applications assume that fonts included in original Windows 3.1 distribution are always present. By default Wine creates a number of aliases that map them on the existing X fonts:

Windows font	...is mapped to...	X font
"MS Sans Serif"	->	"-adobe-helvetica-"
"MS Serif"	->	"-bitstream-charter-"

Windows font	...is mapped to...	X font
"Times New Roman"	->	"-adobe-times-"
"Arial"	->	"-adobe-helvetica-"

There is no default alias for the "System" font. Also, no aliases are created for the fonts that applications install at runtime. The recommended way to deal with this problem is to convert the missing font (see above). If it proves impossible, like in the case with TrueType fonts, you can force the font mapper to choose a closely related X font by adding an alias to the [fonts] section. Make sure that the X font actually exists (with **xfontsel** tool).

```
AliasN = [Windows font], [X font] <, op-
tional "mask X font" flag>
```

Example:

```
Alias0 = System, --international-, subst
Alias1 = ...
...
```

Comments:

- There must be no gaps in the sequence `{0, ..., N}` otherwise all aliases after the first gap won't be read.
- Usually font mapper translates X font names into font names visible to Windows programs in the following fashion:

X font	...will show up as...	Extracted name
--international-...	->	"International"
-adobe-helvetica-...	->	"Helvetica"
-adobe-utopia-...	->	"Utopia"

X font	...will show up as...	Extracted name
-misc-fixed-...	->	"Fixed"
-...	->	
-sony-fixed-...	->	"Sony Fixed"
-...	->	

Note that since `-misc-fixed-` and `-sony-fixed-` are different fonts Wine modified the second extracted name to make sure Windows programs can distinguish them because only extracted names appear in the font selection dialogs.

- "Masking" alias replaces the original extracted name so that in the example case we will have the following mapping:

X font	...is masked to...	Extracted name
--international-...	->	"System"

"Nonmasking" aliases are transparent to the user and they do not replace extracted names.

Wine discards an alias when it sees that the native X font is available.

- If you do not have access to Windows fonts mentioned in the first paragraph you should try to substitute the "System" font with nonmasking alias. The **xfontsel** application will show you the fonts available to X.

```
Alias.. = System, ...bold font without serifs
```

Also, some Windows applications request fonts without specifying the typeface name of the font. Font table starts with Arial in most Windows installations, however X font table starts with whatever is the first line in the `fonts.dir`. Therefore WINE uses the following entry to determine which font to check first.

Example:

```
Default = -adobe-times-
```

Comments:

It is better to have a scalable font family (bolds and italics included) as the default choice because mapper checks all available fonts until requested height and other attributes match perfectly or the end of the font table is reached. Typical X installations have scalable fonts in the `../fonts/Type1` and `../fonts/Speedo` directories.

5.1.3. How To Manage Cached Font Metrics

WINE stores detailed information about available fonts in the `~/.wine/.cachedmetrics` file. You can copy it elsewhere and add this entry to the [fonts] section in your `wine.conf`:

```
FontMetrics = <file with metrics>
```

If WINE detects changes in the X font configuration it will rebuild font metrics from scratch and then it will overwrite `~/.wine/.cachedmetrics` with the new information. This process can take a while.

5.1.4. Too Small Or Too Large Fonts

Windows programs may ask WINE to render a font with the height specified in points. However, point-to-pixel ratio depends on the real physical size of your display (15", 17", etc...). X tries to provide an estimate of that but it can be quite different from the actual size. You can change this ratio by adding the following entry to the [fonts] section:

```
Resolution = <integer value>
```

In general, higher numbers give you larger fonts. Try to experiment with values in the 60 - 120 range. 96 is a good starting point.

5.1.5. "FONT_Init: failed to load ..." Messages On Startup

The most likely cause is a broken `fonts.dir` file in one of your font directories. You need to rerun **mkfontdir** to rebuild this file. Read its manpage for more information. If you can't run **mkfontdir** on this machine as you are not root, use **xset -fp xxx** to remove the broken font path.

5.2. Setting up a TrueType Font Server

written by ???

(Extracted from `wine/documentation/ttfserver`)

Follow these instructions to set up a TrueType font server on your system.

1. Get `freetype-1.0.full.tar.gz`
2. Read docs, unpack, configure and install
3. Test the library, e.g. **ftview 20 /dosc/win95/fonts/times**
4. Get `xfsft-beta1e.linux-i586`
5. Install it and start it when booting, e.g. in an rc-script. The manpage for **xfs** applies.
6. Follow the hints given by `<williamc@dai.ed.ac.uk>`
7. I got **xfsft** from http://www.dcs.ed.ac.uk/home/jec/progindex.html. I have it running all the time. Here is `/usr/X11R6/lib/X11/fs/config`:

```
clone-self = on
use-syslog = off
```

```
catalogue = /c/windows/fonts
error-file = /usr/X11R6/lib/X11/fs/fs-errors
default-point-size = 120
default-resolutions = 75,75,100,100
```

Obviously `/c/windows/fonts` is where my Windows fonts on my Win95 *C:* drive live; could be e.g. `/mnt/dosC/windows/system` for Win31. In `/c/windows/fonts/fonts.scale` I have

```
14
arial.ttf -monotype-arial-medium-r-normal--0-0-0-0-p-0-
iso8859-1
arialbd.ttf -monotype-arial-bold-r-normal--0-0-0-0-p-0-
iso8859-1
arialbi.ttf -monotype-arial-bold-o-normal--0-0-0-0-p-0-
iso8859-1
ariali.ttf -monotype-arial-medium-o-normal--0-0-0-0-p-0-
iso8859-1
cour.ttf -monotype-courier-medium-r-normal--0-0-0-0-p-0-
iso8859-1
courbd.ttf -monotype-courier-bold-r-normal--0-0-0-0-p-0-
iso8859-1
courbi.ttf -monotype-courier-bold-o-normal--0-0-0-0-p-0-
iso8859-1
couri.ttf -monotype-courier-medium-o-normal--0-0-0-0-p-0-
iso8859-1
times.ttf -monotype-times-medium-r-normal--0-0-0-0-p-0-
iso8859-1
timesbd.ttf -monotype-times-bold-r-normal--0-0-0-0-p-0-
iso8859-1
timesbi.ttf -monotype-times-bold-i-normal--0-0-0-0-p-0-
iso8859-1
timesi.ttf -monotype-times-medium-i-normal--0-0-0-0-p-0-
iso8859-1
symbol.ttf -monotype-symbol-medium-r-normal--0-0-0-0-p-0-
microsoft-symbol
```

```
wingding.ttf -microsoft-wingdings-medium-r-normal--0-0-0-0-p-
0-microsoft-symbol
```

In `/c/windows/fonts/fonts.dir` I have exactly the same.

In `/usr/X11R6/lib/X11/XF86Config` I have

```
FontPath "tcp/localhost:7100"
```

in front of the other `FontPath` lines. That's it! As an interesting by-product of course, all those web pages which specify Arial come up in Arial in Netscape ...

8. Shut down X and restart (and debug errors you did while setting up everything).
9. Test with e.g **xlsfont | grep arial**

Hope this helps...

Chapter 6. Printing in Wine

How to print documents in Wine...

6.1. Printing

written by (???)

(Extracted from `wine/documentation/printing`)

Printing in Wine can be done in one of two ways. Both of which are very alpha.

1. Use an external windows 3.1 printer driver.
2. Use the builtin Wine Postscript driver (+ ghostscript to produce output for non-postscript printers).

Note that at the moment WinPrinters (cheap, dumb printers that require the host computer to explicitly control the head) will not work. It is unclear whether they ever will.

6.1.1. External printer drivers

At present only 16 bit drivers will work (note that these include win9x drivers). To use them, add

```
printer=on
```

to the [wine] section of `wine.conf` (or `~/.winerc`). This lets `CreateDC` proceed if its driver argument is a 16 bit driver. You will probably also need to add

```
TTEnable=0 TTOnly=0
```

to the [TrueType] section of `win.ini`. The code for the driver interface is in `graphics/win16drv`.

6.1.2. Builtin Wine PostScript driver

Enables printing of PostScript files via a driver built into Wine. See `documentation/psdriver` for installation instructions. The code for the PostScript driver is in `graphics/psdrv`.

6.1.3. Spooling

Spooling is rather primitive. The [spooler] section of `wine.conf` maps a port (e.g. LPT1:) to a file or a command via a pipe. For example the following lines

```
LPT1:=foo.ps LPT2:=|lpr
```

map LPT1: to file `foo.ps` and LPT2: to the **lpr** command. If a job is sent to an unlisted port then a file is created with that port's name e.g. for LPT3: a file called LPT3: would be created.

6.2. The Wine PostScript Driver

written by Huw Davies <h.davies1@physics.ox.ac.uk>

(Extracted from `wine/documentation/psdriver`)

When complete this will allow Wine to generate PostScript files without needing an external printer driver. It should be possible to print to a non PostScript printer by filtering the output through ghostscript.

6.2.1. Installation

The driver behaves as if it were a DRV file called `wineps.drv` which at the moment is built into Wine. Although it mimics a 16 bit driver it will work with both 16 and 32 bit apps, just as win9x drivers do.

To install it add

```
Wine PostScript Driver=WINEPS,LPT1:
```

to the [devices] section of `win.ini` and to set it as the default printer also add

```
device=Wine PostScript Driver,WINEPS,LPT1:
```

to the [windows] section of `win.ini` and ??? *[sic]*

To run 32 bit apps (and 16 bit apps using the `builtin` commdlg) you also need to add certain entries to the registry. The easiest way to do that at the moment is to use the winelib program **programs/regapi/regapi** with the file `documentation/psdrv.reg`. To do this **cd** to `programs/regapi/regapi` and type **`make`** to actually make the program, then type **`./regapi setValue <../../documentation/psdrv.reg`**. You can obviously edit `psdrv.reg` to suit your requirements.

You will need Adobe Font Metric (AFM) files for the (type 1 PostScript) fonts that you wish to use. You can get these from ftp://ftp.adobe.com/pub/adobe/type/win/all/afmfiles (ftp://ftp.adobe.com/pub/adobe/type/win/all/afmfiles). The directories `base17` or `base35` are good places to start. Note that these are only the font metrics and not the fonts themselves. At the moment the driver does not download additional fonts, so you can only use fonts that are already present on the printer.

Then create a [afmfiles] section in your `wine.conf` (or `~/.winerc`) and add a line of the form

```
file<n>=/unix/path/name/filename.afm
```

for each AFM file that you wish to use. [This might change in the future]

You also require a PPD file for your printer. This describes certain characteristics of the printer such as which fonts are installed, how to select manual feed etc. Adobe also has many of these on its website, have a look in ftp://ftp.adobe.com/pub/adobe/printerdrivers/win/all/ (ftp://ftp.adobe.com/pub/adobe/printerdrivers/win/all/). Create a [psdrv] section in your `wine.conf` (or `~/.winerc`) and add the following entry:

```
ppdfile=/somewhere/file.ppd
```

By default, the driver will look for a file named `default.ppd` in the directory from which you started wine.

To enable colour printing you need to have the `*ColorDevice` entry in the PPD set to `true`, otherwise the driver will generate greyscale.

Note that you need not set `printer=on` in the [wine] section of `wine.conf`, this enables printing via external printer drivers and does not affect wineps.

If you're lucky you should now be able to produce PS files from Wine!

I've tested it with win3.1 notepad/write, Winword6 and Origin4.0 and 32 bit apps such as win98 wordpad, Winword97, Powerpoint2000 with some degree of success - you should be able to get something out, it may not be in the right place.

6.2.2. TODO / Bugs

- Driver does read PPD files, but ignores all constraints and doesn't let you specify whether you have optional extras such as envelope feeders. You will therefore find a larger than normal selection of input bins in the print setup dialog box. I've only really tested ppd parsing on the `hp4m6_v1.ppd` file.

- No TrueType download.
- StretchDIBits uses level 2 PostScript.
- AdvancedSetup dialog box.
- Many partially implemented functions.
- ps.c is becoming messy.
- Notepad often starts text too far to the left depending on the margin settings. However the win3.1 `pscript.drv` (under wine) also does this.
- Probably many more...

Please contact me if you want to help so that we can avoid duplication.

Huw Davies <h.davies1@physics.ox.ac.uk>

II. Developing Wine

Table of Contents

red7. Compiling Wine red83
red8. Debugging Wine red84
red9. Documenting Wine red97
red10. Submitting Patches red99
red11. Internationalization red100
red12. Porting Wine to new Platforms red101

Chapter 7. Compiling Wine

How to compile wine, and problems that may arise...

Chapter 8. Debugging Wine

8.1. Debug Messages

written by Dimitrie O. Paun <dimi@cs.toronto.edu>, 28 Mar 1998

(Extracted from wine/documentation/debug-msgs)

Note: The new debugging interface can be considered to be stable, with the exception of the in-memory message construction functions. However, there is still a lot of work to be done to polish things up. To make my life easier, please follow the guidelines described in this document.

Important: Read this document before writing new code. DO NOT USE fprintf (or printf) to output things. Also, instead of writing FIXMEs in the source, output a FIXME message if you can.

At the end of the document, there is a "Style Guide" for debugging messages. Please read it.

8.1.1. Debugging classes

There are 4 types (or classes) of debugging messages:

FIXME

Messages in this class relate to behavior of Wine that does not correspond to standard Windows behavior and that should be fixed.

Examples: stubs, semi-implemented features, etc.

ERR

Messages in this class relate to serious errors in Wine. This sort of messages are close to asserts -- that is, you should output an error message when the code detects a condition which should not happen. In other words, important things that are not warnings (see below), are errors.

Examples: unexpected change in internal state, etc.

WARN

These are warning messages. You should report a warning when something unwanted happen but the function behaves properly. That is, output a warning when you encounter something unexpected (ex: could not open a file) but the function deals correctly with the situation (that is, according to the docs). If you do not deal correctly with it, output a fixme.

Examples: fail to access a resource required by the app, etc.

TRACE

These are detailed debugging messages that are mainly useful to debug a component. These are usually turned off.

Examples: everything else that does not fall in one of the above mentioned categories and the user does not need to know about it.

The user has the capability to turn on or off messages of a particular type. You can expect the following patterns of usage (but note that any combination is possible):

- when you debug a component, all types (TRACE, WARN, ERR, FIXME) will be enabled.
- during the pre-alpha (maybe alpha) stage of Wine, most likely the TRACE class will be disabled by default, but all others (WARN, ERR, FIXME) will be enabled by default.
- when Wine will become stable, most likely the TRACE and WARN classes will be disabled by default, but all ERRs and FIXMEs will be enabled.
- in some installations that want the smallest footprint and where the debug information is of no interest, all classes may be disabled by default.

Of course, the user will have the runtime ability to override these defaults. However, this ability may be turned off and certain classes of messages may be completely disabled at compile time to reduce the size of Wine.

8.1.2. Debugging channels

Also, we divide the debugging messages on a component basis. Each component is assigned a debugging channel. The identifier of the channel must be a valid C identifier but note that it may also be a reserved word like int or static.

Examples of debugging channels: reg, updown, string

We will refer to a generic channel as xxx.

Note: for those who know the old interface, the channel/type is what followed the _ in the dprintf_xxx statements. For example, to output a message on the debugging channel reg in the old interface you would had to write:

In the new interface, we drop the `stddeb` as it is implicit. However, we add an orthogonal piece of information to the message: its class. This is very important as it will allow us to selectively turn on or off certain messages based on the type of information they report. For this reason it is essential to choose the right class for the message. Anyhow, suppose we figured that this message should belong in the `WARN` class, so in the new interface, you write:

```
WARN(reg, "Could not access key!\n");
```

8.1.3. How to use it

So, to output a message (class `YYY`) on channel `xxx`, do:

```
#include "debug.h"

....

YYY(xxx, "<message>", ...);
```

Some examples from the code:

```
#include "debug.h"

...

  TRACE(crtdll, "CRTDLL_setbuf(file %p buf %p)", file, buf);

  WARN(aspi, "Error opening device errno=%d", save_error);
```

If you need to declare a new debugging channel, use it in your code and then do:

```
%tools/make_debug
```

in the root directory of Wine. Note that this will result in almost complete recompilation of Wine.

1. Please pay attention to which class you assign the message. There are only 4 classes, so it is not hard. The reason it is important to get it right is that too much information is no information. For example, if you put things into the `WARN` class that should really be in the `TRACE` class, the output will be too big and this will force the user to turn warnings off. But this way he will fail to see the important ones. Also, if you put warnings into the `TRACE` class lets say, he will most likely miss those because usually the `TRACE` class is turned off. A similar argument can be made if you mix any other two classes.
2. All lines should end with a newline. If you can NOT output everything that you want in the line with only one statement, then you need to build the string in memory. Please read the section below "In-memory messages" on the preferred way to do it. PLEASE USE THAT INTERFACE TO BUILD MESSAGES IN MEMORY. The reason is that we are not sure that we like it and having everything in one format will facilitate the (automatic) translation to a better interface.

8.1.4. Are we debugging?

To test whether the debugging output of class `yyy` on channel `xxx` is enabled, use:

```
TRACE_ON  to test if TRACE is enabled
WARN_ON   to test if WARN is enabled
FIXME_ON  to test if FIXME is enabled
ERR_ON    to test if ERR is enabled
```

Examples:

```
if(TRACE_ON(atom)){
  ...blah...
}
```

Note: You should normally need to test only if `TRACE_ON`. At present, none of the other 3 tests (except for `ERR_ON` which is used only once!) are used in Wine.

8.1.5. In-memory messages

If you NEED to build the message from multiple calls, you need to build it in memory. To do that, you should use the following interface:

1. declare a string (where you are allowed to declare C variables) as follows:
   `dbg_decl_str(name, len);`

where *name* is the name of the string (you should use the channel name on which you are going to output it)

2. print in it with:
   `dsprintf(name, "<message>", ...);`

which is just like a `sprintf` function but instead of a C string as first parameter it takes the name you used to declare it.

3. obtain a pointer to the string with: `dbg_str(name)`
4. reset the string (if you want to reuse it with):
   `dbg_reset_str(name);`

Example (modified from the code):

```
void some_func(tabs)
{
  INT32 i;
  LPINT16 p = (LPINT16)tabs;
  dbg_decl_str(listbox, 256);                    /* declare the string */

  for (i = 0; i < descr->nb_tabs; i++) {
    descr->tabs[i] = *p++<<1;
    if(TRACING(listbox))                          /* write in it only if
      dsprintf(listbox, "%hd ", descr->tabs[i]); /* we are gonna output it */
  }
  TRACE(listbox, "Listbox %04x: settabstops %s",
wnd->hwndSelf, dbg_str(listbox));        /* output the whole thing */
}
```

If you need to use it two times in the same scope do like this:

```
void some_func(tabs)
{
  INT32 i;
  LPINT16 p = (LPINT16)tabs;
  dbg_decl_str(listbox, 256);                    /* declare the string      */

  for (i = 0; i < descr->nb_tabs; i++) {
    descr->tabs[i] = *p++<<1;
    if(TRACING(listbox))                          /* write in it only if
      dsprintf(listbox, "%hd ", descr->tabs[i]); /* we are gonna output it */
  }
  TRACE(listbox, "Listbox %04x: settabstops %s\n",
wnd->hwndSelf, dbg_str(listbox));        /* output the whole thing */

  dbg_reset_str(listbox);                         /* !!!reset the string!!! */
  for (i = 0; i < descr->extrainfo_nr; i++) {
    descr->extrainfo = *p+1;
    if(TRACING(listbox))                          /* write in it only if
      dsprintf(listbox,"%3d ",descr->extrainfo); /* we are gonna output it */
  }

  TRACE(listbox, "Listbox %04x: extrainfo %s\n",
wnd->hwndSelf, dbg_str(listbox));        /* output the whole thing */

}
```

Important: As I already stated, I do not think this will be the ultimate interface for building in-memory debugging messages. In fact, I do have better ideas which I hope to have time to implement for the next release. For this reason, please try not to use it. However, if you need to output a line in more than one

`dprintf_xxx` calls, then USE THIS INTERFACE. DO NOT use other methods. This way, I will easily translate everything to the new interface (when it will become available). So, if you need to use it, then follow the following guidelines:

- wrap calls to `dsprintf` with a

```
if(YYY(xxx))
  dsprintf(xxx,...);
```

  Of course, if the call to `dsprintf` is made from within a function which you know is called only if `YYY(xxx)` is true, for example if you call it only like this:

```
if(YYY(xxx))
  print_some_debug_info();
```

  then you need not (and should not) wrap calls to `dsprintf` with the before mentioned `if`.
- name the string EXACTLY like the debugging channel on which is going to be output. Please see the above example.

8.1.6. Resource identifiers

Resource identifiers can be either strings or numbers. To make life a bit easier for outputting these beasts (and to help you avoid the need to build the message in memory), I introduced a new function called `debugres`.
The function is defined in `debugstr.h` and has the following prototype:

```
LPSTR debugres(const void *id);
```

It takes a pointer to the resource id and returns a nicely formatted string of the identifier. If the high word of the pointer is `0`, then it assumes that the identifier is a number and thus returns a string of the form:

```
#xxxx
```

where `xxxx` are 4 hex-digits representing the low word of *`id`*.
If the high word of the pointer is not `0`, then it assumes that the identifier is a string and thus returns a string of the form:

```
'<identifier>'
```

Thus, to use it, do something on the following lines:

```
#include "debug.h"

...

   YYY(xxx, "resource is %s", debugres(myresource));
```

8.1.7. The *--debugmsg* command line option

So, the *`--debugmsg`* command line option has been changed as follows:

- the new syntax is: *`--debugmsg [yyy]#xxx[,[yyy1]#xxx1]*`* where `#` is either `+` or `-`
- when the optional class argument (`yyy`) is not present, then the statement will enable(+)/disable(-) all messages for the given channel (`xxx`) on all classes. For example:

```
--debugmsg +reg,-file
```

  enables all messages on the `reg` channel and disables all messages on the `file` channel. This is same as the old semantics.
- when the optional class argument (`yyy`) is present, then the statement will enable (+)/disable(-) messages for the given channel (`xxx`) only on the given class. For example:

```
--debugmsg trace+reg,warn-file
```

  enables trace messages on the `reg` channel and disables warning messages on the `file` channel.
- also, the pseudo-channel all is also supported and it has the intuitive semantics:

```
    --debugmsg +all      -- enables all debug messages
    --debugmsg -all      -- disables all debug messages
    --debugmsg yyy+all   -- enables debug messages for class yyy on all
                            channels.
```

```
                                channels.
```

So, for example:

```
    --debugmsg warn-all  -- disables all warning messages.
```

Also, note that at the moment:

- the FIXME and ERR classes are enabled by default
- the TRACE and WARN classes are disabled by default

8.1.8. Compiling Out Debugging Messages

To compile out the debugging messages, provide **configure** with the following options:

```
    --disable-debug      -- turns off TRACE, WARN, and FIXME (and DUMP).
    --disable-trace      -- turns off TRACE only.
```

This will result in an executable that, when stripped, is about 15%-20% smaller. Note, however, that you will not be able to effectively debug Wine without these messages.
This feature has not been extensively tested--it may subtly break some things.

8.1.9. A Few Notes on Style

This new scheme makes certain things more consistent but there is still room for improvement by using a common style of debug messages. Before I continue, let me note that the output format is the following:

```
yyy:xxx:fff <message>

where:
  yyy = the class (fixme, err, warn, trace)
  xxx = the channel (atom, win, font, etc)
  fff = the function name
```

these fields are output automatically. All you have to provide is the <message> part.
So here are some ideas:

- do NOT include the name of the function: it is included automatically
- if you want to output the parameters of the function, do it as the first thing and include them in parentheses, like this:

```
  YYY(xxx, "(%d,%p,etc)...\n", par1, par2, ...);
```

- for stubs, you should output a FIXME message. I suggest this style:

```
    FIXME(xxx, "(%x,%d...): stub\n", par1, par2, ...);
```

That is, you output the parameters, then a : and then a string containing the word "stub". I've seen "empty stub", and others, but I think that just "stub" suffices.

- output 1 and ONLY 1 line per message. That is, the format string should contain only 1 \n and it should always appear at the end of the string. (there are many reasons for this requirement, one of them is that each debug macro adds things to the beginning of the line)
- if you want to name a value, use = and NOT :. That is, instead of saying:

```
  FIXME(xxx, "(fd: %d, file: %s): stub\n", fd, name);
```

say:

```
  FIXME(xxx, "(fd=%d, file=%s): stub\n", fd, name);
```

use : to separate categories.

- try to avoid the style:

```
  FIXME(xxx, "(fd=%d, file=%s): stub\n", fd, name);
```

but use:

```
  FIXME(xxx, "(fd=%d, file=%s): stub\n", fd, name);
```

The reason is that if you want to **grep** for things, you would search for FIXME but in the first case there is no

- if you output a string s that might contain control characters, or if *s* may be NULL, use debugstr_a (for ASCII strings, or debugstr_w for Unicode strings) to convert *s* to a C string, like this:

```
HANDLE32 WINAPI YourFunc(LPCSTR s)
{
    FIXME(xxx, "(%s): stub\n", debugstr_a(s));
}
```

- if you want to output a resource identifier, use debugres to convert it to a string first, like this:

```
HANDLE32 WINAPI YourFunc(LPCSTR res)
{
    FIXME(xxx, "(res=%s): stub\n", debugres(s));
}
```

if the resource identifier is a SEGPTR, use PTR_SEG_TO_LIN to get a liner pointer first:

```
HRSRC16 WINAPI FindResource16( HMODULE16 hModule, SEGPTR name, SEGPTR type )
{
[...]
    TRACE(resource, "module=%04x name=%s type=%s\n",
 hModule, debugres(PTR_SEG_TO_LIN(name)),
 debugres(PTR_SEG_TO_LIN(type)) );
[...]
}
```

- for messages intended for the user (specifically those that report errors in wine.conf), use the MSG macro. Use it like a printf:

```
MSG( "Definition of drive %d is incorrect!\n", drive );
```

However, note that there are *very* few valid uses of this macro. Most messages are debugging messages, so chances are you will not need to use this macro. Grep the source to get an idea where it is appropriate to use it.

- For structure dumps, use the DUMP macro. Use it like a printf, just like the MSG macro. Similarly, there are only a few valid uses of this macro. Grep the source to see when to use it.

8.2. Using the Wine Debugger

written by Marcus Meissner <msmeissn@cip.informatik.uni-erlangen.de>, additions welcome.
(Extracted from wine/documentation/debugging)
This file describes where to start debugging Wine. If at any point you get stuck and want to ask for help, please read the file documentation/bugreports for information on how to write useful bug reports.

8.2.1. Crashes

These usually show up like this:

```
|Unexpected Windows program segfault - opcode = 8b
|Segmentation fault in Windows program 1b7:c41.
|Loading symbols from ELF file /root/wine/wine...
|....more Loading symbols from ...
|In 16 bit mode.
|Register dump:
| CS:01b7 SS:016f DS:0287 ES:0000
| IP:0c41 SP:878a BP:8796 FLAGS:0246
| AX:811e BX:0000 CX:0000 DX:0000 SI:0001 DI:ffff
|Stack dump:
|0x016f:0x878a:  0001 016f ffed 0000 0000 0287 890b 1e5b
|0x016f:0x879a:  01b7 0001 000d 1050 08b7 016f 0001 000d
|0x016f:0x87aa:  000a 0003 0004 0000 0007 0007 0190 0000
|0x016f:0x87ba:
|
|0050: sel=0287 base=40211d30 limit=0b93f (bytes) 16-bit rw-
|Backtrace:
|0 0x01b7:0x0c41 (PXSRV_FONGETFACENAME+0x7c)
```

```
|2 0x01a7:0x05aa
|3 0x01b7:0x0768 (PXSRV_FONINITFONTS+0x81)
|4 0x014f:0x03ed (PDOXWIN_@SQLCURCB$Q6CBTYPEULN8CBSCTYPE+0x1b1)
|5 0x013f:0x00ac
|
|0x01b7:0x0c41 (PXSRV_FONGETFACENAME+0x7c):  movw        %es:0x38(%bx),%dx
```

Steps to debug a crash. You may stop at any step, but please report the bug and provide as much of the information gathered to the newsgroup or the relevant developer as feasible.

1. Get the reason for the crash. This is usually an access to an invalid selector, an access to an out of range address in a valid selector, popping a segmentregister from the stack or the like. When reporting a crash, report this *whole* crashdump even if it doesn't make sense to you.
   (In this case it is access to an invalid selector, for %es is `0000`, as seen in the register dump).
2. Determine the cause of the crash. Since this is usually a primary/secondary reaction to a failed or misbehaving Wine function, rerun Wine with *`-debugmsg +relay`* added to the commandline. This will generate quite a lot of output, but usually the reason is located in the last call(s). Those lines usually look like this:

```
|Call KERNEL.90: LSTRLEN(0227:0692 "text") ret=01e7:2ce7 ds=0227
      ^^^^^^^^^  ^       ^^^^^^^^^ ^^^^^^      ^^^^^^^^^    ^^^^
      |          |       |         |           |            |Datasegment
      |          |       |         |           |Return address
      |          |       |         |textual parameter
      |          |       |
      |          |       |Argument(s). This one is a win16 segmented pointer.
      |          |Function called.
      |The module, the function is called in. In this case it is KERNEL.

|Ret  KERNEL.90: LSTRLEN() retval=0x0004 ret=01e7:2ce7 ds=0227
                                ^^^^^^
  |Returnvalue is 16 bit and has the value 4.
```

3. If you have found a misbehaving function, try to find out why it misbehaves. Find the function in the source code. Try to make sense of the arguments passed. Usually there is a `TRACE(<channel>,"(...)\n");` at the beginning of the function. Rerun wine with *`-debugmsg +xyz,+relay`* added to the commandline.
4. Additional information on how to debug using the internal debugger can be found in `debugger/README`.
5. If this information isn't clear enough or if you want to know more about what's happening in the function itself, try running wine with *`-debugmsg +all`*, which dumps ALL included debug information in wine.
6. If even that isn't enough, add more debug output for yourself into the functions you find relevant. See `documentation/debug-msgs`. You might also try to run the program in **gdb** instead of using the WINE-debugger. If you do that, use *`handle SIGSEGV nostop noprint`* to disable the handling of seg faults inside **gdb** (needed for Win16). If you don't use the *`--desktop`* or *`--managed`* option, start the WINE process with *`--sync`*, or chances are good to get X into an unusable state.
7. You can also set a breakpoint for that function. Start wine with the *`--debug`* option added to the commandline. After loading the executable wine will enter the internal debugger. Use *`break KERNEL_LSTRLEN`* (replace by function you want to debug, CASE IS RELEVANT) to set a breakpoint. Then use **continue** to start normal program-execution. Wine will stop if it reaches the breakpoint. If the program isn't yet at the crashing call of that function, use **continue** again until you are about to enter that function. You may now proceed with single-stepping the function until you reach the point of crash. Use the other debugger commands to print registers and the like.

8.2.2. Program hangs, nothing happens

Switch to UNIX shell, get the process-ID using **ps -a | grep wine**, and do a **kill -HUP <pid>** (without the < and >). Wine will then enter its internal debugger and you can proceed as explained above. Also, you can use *`--debug`* switch and then you can get into internal debugger by pressing **Ctrl-C** in the terminal where you run Wine.

8.2.3. Program reports an error with a Messagebox

Sometimes programs are reporting failure using more or less nondescript messageboxes. We can debug this using the same method as Crashes, but there is one problem... For setting up a message box the program also calls Wine producing huge chunks of debug code.

Since the failure happens usually directly before setting up the Messagebox you can start wine with *--debug* added to the commandline, set a breakpoint at `MessageBoxA` (called by win16 and win32 programs) and proceed with **continue**. With *--debugmsg +all* Wine will now stop directly before setting up the Messagebox. Proceed as explained above.

You can also run wine using **wine -debugmsg +relay program.exe 2>&1 | less -i** and in **less** search for "MessageBox".

8.2.4. Disassembling programs:

You may also try to disassemble the offending program to check for undocumented features and/or use of them. The best, freely available, disassembler for Win16 programs is Windows Codeback, archivename `wcbxxx.zip`, which usually can be found in the `Cica-Mirror` subdirectory on the WINE ftpsites. (See `ANNOUNCE`). Disassembling win32 programs is possible using Windows Disassembler 32, archivename something like `w32dsm87.zip` (or similar) on ftp.winsite.com and mirrors. The shareware version does not allow saving of disassembly listings. You can also use the newer (and in the full version better) Interactive Disassembler (IDA) from the ftp sites mentioned at the end of the document. Understanding disassembled code is mostly a question of exercise.

Most code out there uses standard C function entries (for it is usually written in C). Win16 function entries usually look like that:

```
push bp
mov bp, sp
... function code ..
retf XXXX  <--------- XXXX is number of bytes of arguments
```

This is a `FAR` function with no local storage. The arguments usually start at `[bp+6]` with increasing offsets. Note, that `[bp+6]` belongs to the *rightmost* argument, for exported win16 functions use the PASCAL calling convention. So, if we use `strcmp(a,b)` with *a* and *b* both 32 bit variables *b* would be at `[bp+6]` and *a* at `[bp+10]`.

Most functions make also use of local storage in the stackframe:

```
enter 0086, 00
... function code ...
leave
retf XXXX
```

This does mostly the same as above, but also adds `0x86` bytes of stackstorage, which is accessed using `[bp-xx]`.

Before calling a function, arguments are pushed on the stack using something like this:

```
push word ptr [bp-02] <- will be at [bp+8]
push di <- will be at [bp+6]
call KERNEL.LSTRLEN
```

Here first the selector and then the offset to the passed string are pushed.

8.2.5. Sample debugging session:

Let's debug the infamous Word `SHARE.EXE` messagebox:

```
|marcus@jet $ wine winword.exe
|            +--------------------------------------------+
|            | !  You must leave Windows and load SHARE.EXE|
|            |    before starting Word.                    |
|            +--------------------------------------------+

|marcus@jet $ wine winword.exe -debugmsg +relay -debug
|CallTo32(wndproc=0x40065bc0,hwnd=000001ac,msg=00000081,wp=00000000,lp=00000000)
|Win16 task 'winword': Breakpoint 1 at 0x01d7:0x001a
|CallTo16(func=0127:0070,ds=0927)
|Call WPROCS.24: TASK_RESCHEDULE() ret=00b7:1456 ds=0927
|Ret  WPROCS.24: TASK_RESCHEDULE() retval=0x8672 ret=00b7:1456 ds=0927
```

```
|     AX=0000 BX=3cb4 CX=1f40 DX=0000 SI=0000 DI=0927 BP=0000 ES=11f7
|Loading symbols: /home/marcus/wine/wine...
|Stopped on breakpoint 1 at 0x01d7:0x001a
|In 16 bit mode.
|Wine-dbg>break MessageBoxA                          <---- Set Breakpoint
|Breakpoint 2 at 0x40189100 (MessageBoxA [msgbox.c:190])
|Wine-dbg>c                                           <---- Continue
|Call KERNEL.91: INITTASK() ret=0157:0022 ds=08a7
|     AX=0000 BX=3cb4 CX=1f40 DX=0000 SI=0000 DI=08a7 ES=11d7 EFL=00000286
|CallTo16(func=090f:085c,ds=0dcf,0x0000,0x0000,0x0000,0x0000,0x0800,0x0000,0x0000,0x0dc
|...                                                  <----- Much debugoutput
|Call KERNEL.136: GETDRIVETYPE(0x0000) ret=060f:097b ds=0927
                               ^^^^^^ Drive 0 (A:)
|Ret  KERNEL.136: GETDRIVETYPE() retval=0x0002 ret=060f:097b ds=0927
                                       ^^^^^^  DRIVE_REMOVEABLE
(It is a floppy diskdrive.)

|Call KERNEL.136: GETDRIVETYPE(0x0001) ret=060f:097b ds=0927
                               ^^^^^^ Drive 1 (B:)
|Ret  KERNEL.136: GETDRIVETYPE() retval=0x0000 ret=060f:097b ds=0927
                                       ^^^^^^  DRIVE_CANNOTDETERMINE
(I don't have drive B: assigned)

|Call KERNEL.136: GETDRIVETYPE(0x0002) ret=060f:097b ds=0927
                               ^^^^^^^ Drive 2 (C:)
|Ret  KERNEL.136: GETDRIVETYPE() retval=0x0003 ret=060f:097b ds=0927
                                       ^^^^^^ DRIVE_FIXED
                                              (specified as a harddisk)

|Call KERNEL.97: GETTEMPFILE-
NAME(0x00c3,0x09278364"doc",0x0000,0927:8248) ret=060f:09b1 ds=0927
                                 ^^^^^^           ^^^^^        ^^^^^^^^^
                                 |                |            |buffer for fname
                                 |                |temporary name ~docXXXX.tmp
                                 |Force use of Drive C:.

|Warning: GetTempFileName re-
turns 'C:~doc9281.tmp', which doesn't seem to be writeable.
|Please check your configuration file if this generates a failure.
```

Whoops, it even detects that something is wrong!

```
|Ret  KERNEL.97: GETTEMPFILENAME() retval=0x9281 ret=060f:09b1 ds=0927
                                       ^^^^^^ Temporary storage ID

|Call KERNEL.74: OPEN-
FILE(0x09278248"C:~doc9281.tmp",0927:82da,0x1012) ret=060f:09d8 ds=0927
                                  ^^^^^^^^^^^^^^^^ ^^^^^^^^^ ^^^^^^^
                                  |filename        |OFSTRUCT |open mode:

                                     OF_CREATE|OF_SHARE_EXCLUSIVE|OF_READWRITE
```

This fails, since my *C:* drive is in this case mounted readonly.

```
|Ret  KERNEL.74: OPENFILE() retval=0xffff ret=060f:09d8 ds=0927
                                  ^^^^^^ HFILE_ERROR16, yes, it failed.

|Call USER.1: MESSAGEBOX(0x0000,0x09278376"Sie mussen Windows ver-
lassen und SHARE.EXE laden bevor Sie Word starten.",0x00000000,0x1030) ret=060f:084f ds=
```

And MessageBox'ed.

```
|190     { <- the sourceline
In 32 bit mode.
Wine-dbg>
```

The code seems to find a writeable harddisk and tries to create a file there. To work around this bug, you can define *C:* as a networkdrive, which is ignored by the code above.

8.2.6. Debugging Tips

Here are some useful debugging tips, added by Andreas Mohr:

- If you have a program crashing at such an early loader phase that you can't use the Wine debugger normally, but Wine already executes the program's start code, then you may use a special trick. You should do a
  `wine --debugmsg +relay program`

  to get a listing of the functions the program calls in its start function. Now you do a
  `wine --debug winfile.exe`

  This way, you get into **Wine-dbg**. Now you can set a breakpoint on any function the program calls in the start function and just type **c** to bypass the eventual calls of Winfile to this function until you are finally at the place where this function gets called by the crashing start function. Now you can proceed with your debugging as usual.
- If you try to run a program and it quits after showing an error messagebox, the problem can usually be identified in the return value of one of the functions executed before `MessageBox()`. That's why you should re-run the program with e.g.
  `wine --debugmsg +relay <program name> &>relmsg`

  Then do a **more relmsg** and search for the last occurrence of a call to the string "MESSAGEBOX". This is a line like

```
Call USER.1: MESSAGEBOX(0x0000,0x01ff1246 "Runtime er-
ror 219 at 0004:1056.",0x00000000,0x1010) ret=01f7:2160 ds=01ff
```

  In my example the lines before the call to `MessageBox()` look like that:

```
Call KERNEL.96: FREELIBRARY(0x0347) ret=01cf:1033 ds=01ff
CallTo16(func=033f:0072,ds=01ff,0x0000)
Ret  KERNEL.96: FREELIBRARY() retval=0x0001 ret=01cf:1033 ds=01ff
Call KERNEL.96: FREELIBRARY(0x036f) ret=01cf:1043 ds=01ff
CallTo16(func=0367:0072,ds=01ff,0x0000)
Ret  KERNEL.96: FREELIBRARY() retval=0x0001 ret=01cf:1043 ds=01ff
Call KERNEL.96: FREELIBRARY(0x031f) ret=01cf:105c ds=01ff
CallTo16(func=0317:0072,ds=01ff,0x0000)
Ret  KERNEL.96: FREELIBRARY() retval=0x0001 ret=01cf:105c ds=01ff
Call USER.171: WIN-
HELP(0x02ac,0x01ff05b4 "COMET.HLP",0x0002,0x00000000) ret=01cf:1070 ds=01ff
CallTo16(func=0117:0080,ds=01ff)
Call WPROCS.24: TASK_RESCHEDULE() ret=00a7:0a2d ds=002b
Ret  WPROCS.24: TASK_RESCHEDULE() retval=0x0000 ret=00a7:0a2d ds=002b
Ret  USER.171: WINHELP() retval=0x0001 ret=01cf:1070 ds=01ff
Call KERNEL.96: FREELIBRARY(0x01be) ret=01df:3e29 ds=01ff
Ret  KERNEL.96: FREELIBRARY() retval=0x0000 ret=01df:3e29 ds=01ff
Call KERNEL.52: FREEPROCINSTANCE(0x02cf00ba) ret=01f7:1460 ds=01ff
Ret  KERNEL.52: FREEPROCINSTANCE() retval=0x0001 ret=01f7:1460 ds=01ff
Call USER.1: MESSAGEBOX(0x0000,0x01ff1246 "Runtime er-
ror 219 at 0004:1056.",0x00000000,0x1010) ret=01f7:2160 ds=01ff
```

  I think that the call to `MessageBox()` in this example is *not* caused by a wrong result value of some previously executed function (it's happening quite often like that), but instead the messagebox complains about a runtime error at `0x0004:0x1056`.

  As the segment value of the address is only `4`, I think that that is only an internal program value. But the offset address reveals something quite interesting: Offset `1056` is very close to the return address of `FREELIBRARY()`:

```
Call KERNEL.96: FREELIBRARY(0x031f) ret=01cf:105c ds=01ff
                                                ^^^^
```

Provided that segment `0x0004` is indeed segment `0x1cf`, we now we can use IDA (available at ftp://ftp.uni-koeln.de/pc/msdos/programming/assembler/ida35bx.zip (ftp://ftp.uni-koeln.de/pc/msdos/programming/assembler/ida35bx.zip)) to disassemble the part that caused the error. We just have to find the address of the call to `FreeLibrary()`. Some lines before that the runtime error occurred. But be careful! In some cases you don't have to disassemble the main program, but instead some DLL called by it in order to find the correct place where the runtime error occurred. That can be determined by finding the origin of the segment value (in this case `0x1cf`).

- If you have created a relay file of some crashing program and want to set a breakpoint at a certain location which is not yet available as the program loads the breakpoint's segment during execution, you may set a breakpoint to `GetVersion16/32` as those functions are called very often.
  Then do a **c** until you are able to set this breakpoint without error message.
- Some useful programs:
  IDA: `ftp://ftp.uni-koeln.de/pc/msdos/programming/assembler/ida35bx.zip (ftp://ftp.uni-koeln.de/pc/msdos/programming/assembler/ida35bx.zip)`
  Very good DOS disassembler ! It's badly needed for debugging Wine sometimes.
  XRAY:
  `ftp://ftp.th-darmstadt.de/pub/machines/ms-dos/SimTel/msdos/asmutil/xray15.zip (ftp://ftp.th-darmstadt.de/pub/machines/ms-dos/SimTel/msdos/asmutil/xray15.zip)`
  Traces DOS calls (Int 21h, DPMI, ...). Use it with Windows to correct file management problems etc.
  pedump: `http://oak.oakland.edu/pub/simtelnet/win95/prog/pedump.zip (http://oak.oakland.edu/pub/simtelnet/win95/prog/pedump.zip)`
  Dumps the imports and exports of a PE (Portable Executable) DLL.

8.2.7. Some basic debugger usages:

After starting your program with

```
wine -debug myprog.exe
```

the program loads and you get a prompt at the program starting point. Then you can set breakpoints:

```
b RoutineName       (by outine name) OR
b *0x812575         (by address)
```

Then you hit **c** (continue) to run the program. It stops at the breakpoint. You can type

```
step                (to step one line) OR
stepi               (to step one machine instruction at a time;
                     here, it helps to know the basic 386
                     instruction set)
info reg            (to see registers)
info stack          (to see hex values in the stack)
info local          (to see local variables)
list <line number>  (to list source code)
x <variable name>   (to examine a variable; only works if code
                     is not compiled with optimization)
x 0x4269978         (to examine a memory location)
?                   (help)
q                   (quit)
```

By hitting **Enter**, you repeat the last command.

8.3. How to do regression testing using Cvs

written by Gerard Patel (???)
(Extracted from `wine/documentation/bugreports`)
A problem that can happen sometimes is 'it used to work before, now it doesn't anymore...'. Here is a step by step procedure to try to pinpoint when the problem occured. This is *NOT* for casual users.

1. Get the 'full cvs' archive from winehq. This archive is the cvs tree but with the tags controling the versioning system. It's a big file (> 15 meg) with a name like full-cvs-<last update date> (it's more than 100mb when uncompressed, you can't very well do this with small, old computers or slow Internet connections).
2. untar it into a repository directory:
```
cd /home/gerard
tar -zxffull-cvs-2000-05-20.tar.gz
mv wine repository
```

3. extract a new destination directory. This directory must not be in a subdirectory of the repository else **cvs** will think it's part of the repository and deny you an extraction in the repository:
```
cd /home/gerard
mv wine wine_current (-> this protects your current wine sandbox, if any)
export CVSROOT=/home/gerard/repository
cd /home/gerard
cvs -d $CVSROOT checkout wine
```

   Note that it's not possible to do a checkout at a given date; you always do the checkout for the last date where the full-cvs-xxx snapshot was generated.
4. you will have now in the `~/wine` directory an image of the cvs tree, on the client side. Now update this image to the date you want:
```
cd /home/gerard/wine
cvs -d $CVSROOT update -D "1999-06-01"
```

   The date format is `YYYY-MM-DD`.
   Many messages will inform you that more recent files have been deleted to set back the client cvs tree to the date you asked, for example:
```
cvs update: tsx11/ts_xf86dga2.c is no longer in the repository
```

   cvs update is not limited to upgrade to a *newer* version as I have believed for far too long :-(
5. Now proceed as for a normal update:
```
./configure
make depend && make
```

   When you have found the exact date when a bug was added to the cvs tree, use something like :
```
cvs -d $CVSROOT diff -D "1999-07-10" -D "1999-07-12"
```

   to get all the differences between the last cvs tree version known to work and code that first displayed the misbehavior.

   Note: I did not include flags for **diff** since they are in my `.cvsrc` file:
```
cvs -z 3
update -dPA
diff -u
```

   From this diff file, particularly the file names, and the `ChangeLog`, it's usually possible to find the different individual patches that were done at this time.
   If any non-programmer reads this, the fastest method to get at the point where the problem occured is to use a binary search, that is, if the problem occured in 1999, start at mid-year, then is the problem is already here, back to 1st April, if not, to 1st October, and so on.
6. The next step is to start from the last working version and to dig the individual contributions from http://www.integrita.com/cgi-local/lwgate.pl/WINE-PATCHES/ (http://www.integrita.com/cgi-local/lwgate.pl/WINE-PATCHES/) (where the Wine patches mailing list is archived)
   If the patch was done by the Wine maintainer or if it was sent directly to his mail address without going first through wine-patches (mailto:wine-patches@winehq.com), you are out of luck as you will never find the patch in the archive. If it is, it's often possible to apply the patches one by one to last working cvs snapshot, compile and test. If you have saved the next candidate as `/home/gerard/buggedpatch1.txt`:

```
patch -p 0 < /home/gerard/buggedpatch1.txt
```

Beware that the committed patch is not always identical to the patch that the author sent to wine-patches, as sometimes the Wine maintainer changes things a bit.
If you find one patch that is getting the cvs source tree to reproduce the problem, you have almost won; post the problem on comp.emulators.windows.wine and there is a chance that the author will jump in to suggest a fix; or there is always the possibility to look hard at the patch until it is coerced to reveal where is the bug :-)

Chapter 9. Documenting Wine

How to help out with the Wine documentation effort...

9.1. Writing Wine API Documentation

written by (???)

(Extracted from `wine/documentation/README.documentation`)

To improve the documentation of the Wine API, just add comments to the existing source. For example,

```
/***********************************************************
 *         CopyMetaFile32A   (GDI32.23)
 *
 *  Copies the metafile corresponding to hSrcMetaFile to either
 *  a disk file, if a filename is given, or to a new memory based
 *  metafile, if lpFileName is NULL.
 *
 * RETURNS
 *
 *  Handle to metafile copy on success, NULL on failure.
 *
 * BUGS
 *
 *  Copying to disk returns NULL even if successful.
 */
HMETAFILE32 WINAPI CopyMetaFile32A(
   HMETAFILE32 hSrcMetaFile, /* handle of metafile to copy */
   LPCSTR lpFilename /* filename if copying to a file */
) { ... }
```

becomes, after processing with **c2man** and **nroff -man**,

```
CopyMetaFileA(3w)                                   CopyMetaFileA(3w)


NAME
       CopyMetaFileA - CopyMetaFile32A   (GDI32.23)

SYNOPSIS
       HMETAFILE32 CopyMetaFileA
       (
            HMETAFILE32 hSrcMetaFile,
            LPCSTR lpFilename
       );

PARAMETERS
       HMETAFILE32 hSrcMetaFile
              Handle of metafile to copy.

       LPCSTR lpFilename
              Filename if copying to a file.

DESCRIPTION
       Copies  the  metafile  corresponding  to  hSrcMetaFile  to
       either a disk file, if a filename is given, or  to  a  new
       memory based metafile, if lpFileName is NULL.

RETURNS
       Handle to metafile copy on success, NULL on failure.

BUGS
       Copying to disk returns NULL even if successful.
```

```
SEE ALSO
       GetMetaFileA(3w),   GetMetaFileW(3w),   CopyMetaFileW(3w),
       PlayMetaFile(3w),  SetMetaFileBitsEx(3w),  GetMetaFileBit-
       sEx(3w)
```

Chapter 10. Submitting Patches

How to create patches, and where to send them...

Chapter 11. Internationalization

11.1. Adding New Languages

written by Morten Welinder, January 1996.

- Thereafter revised Februari 1999 by Klaas van Gend
- Revised again May 23, 1999, Klaas van Gend
- Updated May 26, 2000, Zoran Dzelajlija

(Extracted from `wine/documentation/languages`)

This file documents the necessary procedure for adding a new language to the list of languages that Wine can display system menus and forms in. Currently at least the following languages are still missing:

Bulgarian	Chinese	Greek	Icelandic	Japanese
Romanian	Croatian	Turkish	Slovanian	

> **Note:** *I hope I got all the places where changes are needed. If you see any place missing from the list, submit a patch to this file please. Also note that re-organization of the source code might change the list of places.*

To add a new language you need to be able to translate the relatively few texts, of course. You will need very little knowledge of programming, so you have almost no excuses for not adding your language, right? We should easily be able to support 20 languages within a few months, get going! Apart from re-compilation it'll take you about an hour or two.

To add a new language to the list of languages that Wine can handle you must...

1. Find the language ID in `include/winnls.h`.
2. Look in `ole/ole2nls.c` if your language is already incorporated in the `static const struct NLS_langlocale`. If not: find the appropriate entries in `include/winnls.h` and add them to the list.
3. Edit the parameters defined in `ole/nls/*.nls` to fit your local habits and language.
4. Edit `documentation/wine.man.in` (search for *`-language`*) to show the new language abbreviation.
5. Edit `misc/main.c` variable `Languages` to contain the new language abbreviation and language ID. Also edit struct option_table in `misc/options.c` to show the new abbreviation.
6. Edit `include/options.h` enum `WINE_LANGUAGE` to have a member called `LANG_XX` where `XX` is the new abbreviation.
7. Create a new file `dlls/commdlg/cdlg_XX.rc` (where `XX` is your language abbreviation) containing all menus. Your best bet is to copy `cdlg_En.rc` and start translating. There is no real need to know how the internal structure of the file, as you only need to translate the text within quotes.
   In menus, the character "&" means that the next character will be highlighted and that pressing that letter will select the item. You should place these "&" characters suitably for your language, not just copy the positions from (say) English. In particular, items within one menu should have different highlighted letters.
8. Edit `dlls/commdlg/rsrc.rc` to contain an #include statement for your `cdlg_XX.rc` file.
9. Repeat steps 6 and 7 again for:
   - `dlls/shell32/shell32_XX.rc` and `shres.rc`
   - `resources/sysres_XX.rc` and `user32.rc`
10. Re-configure, re-make dependencies, and re-make Wine.
11. Check your new menus and forms; when they're not ok, go back to 6) and adapt the sizes, etc.
12. Several of the winelib based programs in the subdirectory programs also have internationalisation support. See the appropriate files there for reference.
13. Edit `documentation/internationalisation` to show the new status.
14. Submit patches for inclusion in the next Wine release, see file `./ANNOUNCE` for details about where to submit.

Chapter 12. Porting Wine to new Platforms

Porting Wine to different (UNIX-based) operating systems...

12.1. Porting

written by ???

(Extracted from `wine/documentation/how-to-port`)

12.1.1. What is this?

This note is a short description of:

- How to port Wine to your favourite operating system
- Why you probably shouldn't use #ifdef MyOS
- What to do instead.

This document does not say a thing about how to port Wine to non-386 operating systems, though. You would need a CPU emulator. Let's get Wine into a better shape on 386 first, OK?

12.1.2. Why #ifdef MyOS is probably a mistake.

Operating systems change. Maybe yours doesn't have the `foo.h` header, but maybe a future version will have it. If you want to #include <foo.h>, it doesn't matter what operating system you are using; it only matters whether `foo.h` is there.

Furthermore, operating systems change names or "fork" into several ones. An #ifdef MyOs will break over time.

If you use the feature of **autoconf** -- the Gnu auto-configuration utility -- wisely, you will help future porters automatically because your changes will test for *features*, not names of operating systems. A feature can be many things:

- existance of a header file
- existance of a library function
- existance of libraries
- bugs in header files, library functions, the compiler, ...
- (you name it)

You will need Gnu Autoconf, which you can get from your friendly Gnu mirror. This program takes Wine's `configure.in` file and produces a `configure` shell script that users use to configure Wine to their system.

There *are* exceptions to the "avoid #ifdef MyOS" rule. Wine, for example, needs the internals of the signal stack -- that cannot easily be described in terms of features.

Let's now turn to specific porting problems and how to solve them.

12.1.3. MyOS doesn't have the `foo.h` header!

This first step is to make **autoconf** check for this header. In `configure.in` you add a segment like this in the section that checks for header files (search for "header files"):

```
AC_CHECK_HEADER(foo.h, AC_DEFINE(HAVE_FOO_H))
```

If your operating system supports a header file with the same contents but a different name, say `bar.h`, add a check for that also.

Now you can change

```
#include <foo.h>
```

to

```
#ifdef HAVE_FOO_H
#include <foo.h>
#elif defined (HAVE_BAR_H)
#include <bar.h>
#endif
```

If your system doesn't have a corresponding header file even though it has the library functions being used, you might have to add an #else section to the conditional. Avoid this if you can.

You will also need to add #undef HAVE_FOO_H (etc.) to `include/config.h.in`

Finish up with **make configure** and **./configure**.

12.1.4. MyOS doesn't have the `bar` function!

A typical example of this is the `memmove` function. To solve this problem you would add `memmove` to the list of functions that **autoconf** checks for. In `configure.in` you search for `AC_CHECK_FUNCS` and add `memmove`. (You will notice that someone already did this for this particular function.)

Secondly, you will also need to add #undef HAVE_BAR to `include/config.h.in`

The next step depends on the nature of the missing function.

Case 1:

It's easy to write a complete implementation of the function. (`memmove` belongs to this case.)

You add your implementation in `misc/port.c` surrounded by #ifndef HAVE_MEMMOVE and #endif.

You might have to add a prototype for your function. If so, `include/miscemu.h` might be the place. Don't forget to protect that definition by #ifndef HAVE_MEMMOVE and #endif also!

Case 2:

A general implementation is hard, but Wine is only using a special case.

An example is the various `wait` calls used in `SIGNAL_child` from `loader/signal.c`. Here we have a multi-branch case on features:

```
#ifdef HAVE_THIS
...
#elif defined (HAVE_THAT)
...
#elif defined (HAVE_SOMETHING_ELSE)
...
#endif
```

Note that this is very different from testing on operating systems. If a new version of your operating systems comes out and adds a new function, this code will magically start using it.

Finish up with **make configure** and **./configure**.

12.2. Running & Compiling WINE in OS/2

written by Robert Pouliot <krynos@clic.net>, January 9, 1997

(Extracted from `wine/documentation/wine_os2`)

If you want to help the port of WINE to OS/2, send me a message at <`krynos@clic.net`> I currently don't want beta testers. It must work before we can test it.

Here is what you need to (try to) compile Wine for OS/2:

- EMX 0.9c (fix 2)
- XFree86 3.2 OS/2 (with development libraries)
- **bash**, gnu **make**, **grep**, **tar**, **bison**, **flex**
- **sed** (a working copy of)
- xpm
- **diff** and **patch** are recommended
- Lots of disk space (about 40-50 megs after EMX and XFree installed)

To compile:

```
$ sh
$ tools/make_os2.sh
$ make depend
$ make
$ emxbind wine
```

Currently:

- **configure** and **make depend** work...
- **make** compiles (with a modified Linux `mman.h`), but doesn't link.
- signal handling is horrible... (if any)
- EMX doesn't support `mmap` (and related), SysV IPC and `stafs()`
- XFree86/OS2 3.2 doesn't support `XShmQueryExtension()` and `XShmPixmapFormat()` due to the same lack in EMX...

What needs to be redone:

- LDT (using `DosAllocSeg` in `memory/ldt.c`) *

- File functions,
- I/O access (do it!),
- Communication (modem),
- Interrupt (if int unknown, call current RealMode one...),
- Verify that everything is thread safe (how does Win95/NT handle multi-thread?),
- Move X functions in some files (and make a wrapper, to use PM instead latter),
- Return right CPU type,
- Make winsock work

The good things:

- OS/2 have DOS interrupts
- OS/2 have I/O port access
- OS/2 have multi-thread
- Merlin have Open32 (to be used later...)

III. Distributing Wine

Table of Contents

red13. Packaging Wine red105

Chapter 13. Packaging Wine

13.1. A Small WINE Distribution Guide

written by Marcus Meissner <Marcus.Meissner@caldera.de>

(Extracted from `wine/documentation/distributors`)

While packaging WINE for one of the Linux distributions I came across several points which have not been clarified yet. Particularly a how-to for WINE packaging distributors is missing. This document tries to give a brief overview over the rationales I thought up and how I tried to implement it. (While the examples use **rpm** most of this stuff can be applied to other packagers too.)

Note: YOU SHOULD RECHECK THIS FILE EVERY TWO MONTHS OR SO (**diff -uN** comes to my mind here...). We'll be adding stuff constantly here in order to improve the Wine environment !

1. Rationales

   A WINE install should:

   - Not have a world writeable directory (-tree).
   - Require only as much user input as needed. It would be very good if it would not require any at all. Just let the system administrator do **rpm -i wine.rpm** and let any user be able to run **wine sol.exe** instantly.
   - Give the user as much flexibility as possible to install his own applications, do his own configuring etc.
   - Come as preconfigured as possible, so the user does not need to change any configuration files.
   - Use only as much diskspace as needed per user.

   A WINE install needs:

   - A writeable `C:\` directory structure on a per user basis. Applications do dump `.ini` files into `c:\windows`, installers dump `.exe`, `.dll` and more into `c:\windows\` and subdirectories or into `C:\Program Files\`.
   - The `.exe` and `.dll` from a global read-only Windows installation to be found by applications.
   - Some special `.dll` and `.exe` files in the `windows\system` directory, since applications directly check for their presence.
   - Some special program environment.

2. Implementation

   b.a. Building the package

   WINE is configured the usual way (depending on your build environment). The "prefix" is chosen using your application placement policy (`/usr/`, `/usr/X11R6/`, `/opt/wine/` or similar). The configuration files (`wine.conf`, `wine.userreg`, `wine.systemreg`) are targeted for `/etc/wine/` (rationale: FHS 2.0, multiple readonly configuration files of a package).

   Example (split this into `%build` and `%install` section for **rpm**):

```
CFLAGS=$RPM_OPT_FLAGS \
./configure --prefix=/usr/X11R6 --sysconfdir=/etc/wine/ --enable-dll
make
BR=$RPM_BUILD_ROOT
make install prefix=$BR/usr/X11R6/ sysconfdir=$BR/etc/wine/
install -d $BR/etc/wine/
install -m 644 wine.ini $BR/etc/wine/wine.conf

# Put all our dlls in a seperate directory. (this works only if
# you have a buildroot)
install -d $BR/usr/X11R6/lib/wine
mv $BR/usr/X11R6/lib/lib* $BR/usr/X11R6/lib/wine/

# the clipboard server is started on demand.
install -m 755 windows/x11drv/wineclipsrv $BR/usr/X11R6/bin/

# The WINE server is needed.
install -m 755 server/wineserver $BR/usr/X11R6/bin/
```

   Here we unfortunately do need to create `wineuser.reg` and `winesystem.reg` from the WINE distributed `winedefault.reg`. This can be done using **./regapi** once for one example user and then reusing his `.wine/user.reg` and `.wine/system.reg` files.

   FIXME: this needs to be done better

```
install -m 644 wine.sytemreg $BR/etc/wine/
install -m 644 wine.userreg $BR/etc/wine/
```

There are now a lot of libraries generated by the build process, so a seperate library directory should be used.

```
install -d 755 $BR/usr/X11R6/lib/
mv $BR/
```

You will need to package the files:

```
$prefix/bin/wine, $prefix/bin/dosmod, $prefix/lib/wine/*
$prefix/man/man1/wine.1, $prefix/include/wine/*,
$prefix/bin/wineserver, $prefix/bin/wineclipsrv

%config /etc/wine/*
%doc ... choose from the toplevel directory and documentation/
```

The post-install script:

```
if ! grep -q /usr/X11R6/lib/wine /etc/ld.so.conf; then
    echo "/usr/X11R6/lib/wine" >> /etc/ld.so.conf
fi
/sbin/ldconfig
```

The post-uninstall script:

```
if [ "$1" = 0 ]; then
    perl -ni -e 'print unless m:/usr/X11R6/lib/wine:;' /etc/ld.so.conf
fi
/sbin/ldconfig
```

b.b. Creating a good default configuration file

For the rationales of needing as less input from the user as possible arises the need for a very good configuration file. The one supplied with WINE is currently lacking. We need:

- [Drive X]:
  - A for the floppy. Specify your distributions default floppy mountpoint here.
    `Path=/auto/floppy`
  - C for the `C:\` directory. Here we use the users homedirectory, for most applications do see `C:\` as root-writeable directory of every windows installation and this basically is it in the UNIX-user context.
    `Path=${HOME}`
  - R for the CD-Rom drive. Specify your distributions default CD-ROM drives mountpoint here.
    `Path=/auto/cdrom`
  - T for temporary storage. We do use `/tmp/` (rationale: between process temporary data belongs to `/tmp/`, FHS 2.0)
  - W for the original Windows installation. This drive points to the `windows\` subdirectory of the original windows installation. This avoids problems with renamed `windows` directories (as for instance `lose95`, `win` or `sys\win95`). During compile/package/install we leave this to be `/`, it has to be configured after the package install.
  - Z for the UNIX Root directory. This avoids any problems with "could not find drive for current directory" users occasionaly complain about in the newsgroup and the ircchannel. It also makes the whole directory structure browseable. The type of Z should be network, so applications expect it to be readonly.
    `Path=/`
- [wine]:
    `Windows=c:\windows\` (the windows/ subdirectory in the users

```
  homedirectory)
System=c:\windows\system\ (the windows/system subdirectory in the users
  homedirectory)
Path=c:\windows;c:\windows\system;c:\windows\system32;w:\;w:\system;w:\system
; Using this trick we have in fact two windows installations in one, we
; get the stuff from the readonly installation and can write to our own.
Temp=t:\ (the TEMP directory)
```

- [Tweak.Layout]
  `WineLook=win95 (just the coolest look ;)`

- Possibly modify the [spooler], [serialports] and [parallelports] sections.
  FIXME: possibly more, including printer stuff.

Add this prepared configuration file to the package.

b.c. Installing WINE for the system administrator

Install the package using the usual packager **rpm -i wine.rpm**. You may edit `/etc/wine/wine.conf`, [Drive W], to point to a possible windows installation right after the install. That's it.

Note that on Linux you should somehow try to add the `unhide` mount option (see **man mount**) to the CD-ROM entry in `/etc/fstab` during package install, as several stupid Windows programs mark some setup (!) files as hidden (ISO9660) on CD-ROMs, which will greatly confuse users as they won't find their setup files on the CD-ROMs as they were used on Windows systems when `unhide` is not set ;-\ And of course the setup program will complain that `setup.ins` or some other mess is missing... If you choose to do so, then please make this change verbose to the admin.

b.d. Installing WINE for the user

The user will need to run a setup script before the first invocation of WINE. This script should:

- Copy `/etc/wine/wine.conf` for user modification.
- Allow specification of the original windows installation to use (which modifies the copied `wine.conf` file).
- Create the windows directory structure (`c:\windows`, `c:\windows\system`, `c:\windows\Start Menu\Programs`, `c:\Program Files`, `c:\Desktop`, etc.)
- Symlink all `.dll` and `.exe` files from the original windows installation to the `windows` directory.
  Why? Some programs reference "%windowsdir%/file.dll" or "%systemdir%/file.dll" directly and fail if they are not present.
  This will give a huge number of symlinks, yes. However, if an installer later overwrites on of those files, it will overwrite the symlink (so that the file now lies in the `windows/` subdirectory).
  FIXME: Not sure this is needed for all files.
- On later invocation the script might want to compare regular files in the users windows directories and in the global windows directories and replace same files by symlinks (to avoid diskspace problems).

Done.

This procedure requires:

- Much thought and work from the packager (1x)
- No work for the sysadmin. Well except one **rpm -i** and possible one edit of the configuration file.
- Some or no work from the user, except running the per-user setup script once.
- It scales well and suffices most of the rationales.

Sample `wine.ini` for OpenLinux 2.x:

```
;;
;; MS-DOS drives configuration
;;
;; Each section has the following format:
;; [Drive X]
;; Path=xxx       (Unix path for drive root)
;; Type=xxx       (supported types are 'floppy', 'hd', 'cdrom' and 'network')
;; Label=xxx      (drive label, at most 11 characters)
;; Serial=xxx     (serial number, 8 characters hexadecimal number)
;; Filesystem=xxx (supported types are 'msdos'/'dos'/'fat', 'win95'/'vfat', 'unix')
```

```
;;   This is the FS Wine is supposed to emulate on a certain
;;   directory structure.
;;   Recommended:
;;   - "win95" for ext2fs, VFAT and FAT32
;;   - "msdos" for FAT16 (ugly, upgrading to VFAT driver strongly recommended)
;;   DON'T use "unix" unless you intend to port programs using Winelib !
;; Device=/dev/xx (only if you want to allow raw device access)
;;

;
;
; Floppy 'A' and 'B'
;
; OpenLinux uses an automounter under /auto/, so we use that too.
;
[Drive A]
Path=/auto/floppy/
Type=floppy
Label=Floppy
Serial=87654321
Device=/dev/fd0
Filesystem=win95

;
; Comment in ONLY if you have a second floppy or the automounter hangs
; for 5 minutes.
;
;[Drive B]
;Path=/auto/floppy2/
;Type=floppy
;Label=Floppy
;Serial=87654321
;Device=/dev/fd1
;Filesystem=win95



;
; Drive 'C' links to the users homedirectory.
;
; This must point to a writeable directory structure (not your readonly
; mounted DOS partitions!) since programs want to dump stuff into
; "Program Files/" "Programme/", "windows/", "windows/system/" etc.
;
; The basic structure is set up using the config script.
;
[Drive C]
Path=${HOME}
Type=hd
Label=MS-DOS
Filesystem=win95

;
; /tmp/ directory
;
; The temp drive (and directory) points to /tmp/. Windows programs fill it
; with junk, so it is approbiate.
;
[Drive T]
Path=/tmp
```

```
Label=Tmp Drive
Filesystem=win95

;
; 'U'ser homedirectory
;
; Just in case you want C:\ elsewhere.
;
[Drive U]
Path=${HOME}
Type=hd
Label=Home
Filesystem=win95

;
; CD-'R'OM drive (automounted)
;
; The default cdrom drive.
;
; If an application (or game) wants a specific CD-ROM you might have to
; temporary change the Label to the one of the CD itself.
;
; How to read them is described in /usr/doc/wine-cvs-xxxxx/cdrom-labels.
;
[Drive R]
Path=/auto/cdrom
Type=cdrom
Label=CD-Rom
Filesystem=win95

;
; The drive where the old windows installation resides (it points to the
; windows/ subdirectory).
;
; The Path is modified by the winesetup script.
;
[Drive W]
Path=/
Type=network
Label=Windows
Filesystem=win95
;
; The UNIX Root directory, so all other programs and directories are reachable.
;
; type network is used to tell programs to not write here.
;
[Drive Z]
Path=/
Type=network
Label=ROOT
Filesystem=win95

;
; Standard Windows path entries. WINE will not work if they are incorrect.
;
[wine]
;
; The windows/ directory. It must be writeable, for programs write into it.
;
```

```
;
; The windows/system/ directory. It must be writeable, for especially setup
; programs install dlls in there.
;
System=c:\windows\system
;
; The temp directory. Should be cleaned regulary, since install programs leave
; junk without end in there.
;
Temp=t:\
;
; The dll search path. It should contain at least:
; - the windows and the windows/system directory of the user.
; - the global windows and windows/system directory (from a possible readonly
;   windows installation either on msdos filesystems or somewhere in the UNIX
;   directory tree)
; - any other windows style directories you want to add.
;
Path=c:\windows;c:\windows\system;c:\windows\system32;t:\;w:\;w:\system;w:\system32
;
; Outdated and no longer used. (but needs to be present).
;
SymbolTableFile=./wine.sym

# <wineconf>

;
; Dll loadorder defaults. No need to modify.
;
[DllDefaults]
EXTRA_LD_LIBRARY_PATH=${HOME}/wine/cvs/lib
DefaultLoadOrder = native, elfdll, so, builtin

;
; What 32/16 dlls belong to each other (context wise). No need to modify.
;
[DllPairs]
kernel = kernel32
gdi = gdi32
user = user32
commdlg = comdlg32
commctrl= comctl32
ver = version
shell = shell32
lzexpand= lz32
mmsystem= winmm
msvideo = msvfw32
winsock = wsock32

;
; What type of dll to use in their respective loadorder.
;
[DllOverrides]
kernel32, gdi32, user32 = builtin
kernel, gdi, user = builtin
toolhelp = builtin
comdlg32, commdlg = elfdll, builtin, native
version, ver = elfdll, builtin, native
shell32, shell = builtin, native
```

```
commctrl, comctl32 = builtin, native
wsock32, winsock = builtin
advapi32, crtdll, ntdll = builtin, native
mpr, winspool = builtin, native
ddraw, dinput, dsound = builtin, native
winmm, mmsystem         = builtin
msvideo, msvfw32        = builtin, native
mcicda.drv, mciseq.drv  = builtin, native
mciwave.drv = builtin, native
mciavi.drv, mcianim.drv = native, builtin
w32skrnl                = builtin
wnaspi32, wow32 = builtin
system, display, wprocs = builtin
wineps = builtin

;
; Options section. Does not need to be edited.
;
[options]
; allocate how much system colors on startup. No need to modify.
AllocSystemColors=100

;;
; Font specification. You usually do not need to edit this section.
;
; Read documentation/fonts before adding aliases
;
[fonts]
; The resolution defines what fonts to use (usually either 75 or 100 dpi fonts,
; or nearest match).
Resolution = 96
; Default font
Default = -adobe-times-

;
; serial ports used by "COM1" "COM2" "COM3" "COM4". Useful for applications
; that try to access serial ports.
;
[serialports]
Com1=/dev/ttyS0
Com2=/dev/ttyS1
Com3=/dev/modem,38400
Com4=/dev/modem

;
; parallel port(s) used by "LPT1" etc. Useful for applications that try to
; access these ports.
;
[parallelports]
Lpt1=/dev/lp0

;
; What spooling program to use on printing.
; Use "|program" or "filename", where the output will be dumped into.
;
[spooler]
LPT1:=|lpr
LPT2:=|gs -sDEVICE=bj200 -sOutputFile=/tmp/fred -q -
LPT3:=/dev/lp3
```

```
;
; Allow port access to WINE started by the root user. Useful for some
; supported devices, but it can make the system unstable.
; Read /usr/doc/wine-cvs-xxxxx/ioport-trace-hints.
;
[ports]
;read=0x779,0x379,0x280-0x2a0
;write=0x779,0x379,0x280-0x2a0

; debugging, not need to be modified.
[spy]
Exclude=WM_SIZE;WM_TIMER;

;
; What names for the registry datafiles, no need to modify.
;
[Registry]
; Paths must be given in /dir/dir/file.reg format.
; Wine will not understand dos file names here...
;UserFileName=xxx ; alternate registry file name (user.reg)
;LocalMachineFileName=xxx ; (system.reg)

;
; Layout/Look modifications. Here you can switch with a single line between
; windows 3.1 and windows 95 style.
; This does not change WINE behaviour or reported versions, just the look!
;
[Tweak.Layout]
;; WineLook=xxx  (supported styles are 'Win31'(default), 'Win95', 'Win98')
WineLook=Win95

;
; What programs to start on WINE startup. (you should probably leave it empty)
;
[programs]
Default=
Startup=

; defunct section.
[Console]
;XtermProg=nxterm
;InitialRows=25
;InitialColumns=80
;TerminalType=nxterm

# </wineconf>
```

IV. Wine Architecture

Table of Contents

red14. Overview red114
red15. The Wine Debugger red122
red16. Consoles in Wine red129
red17. Low-level Implementation red131
red18. Wine and OpenGL red137
red19. The Wine Build System red140
red20. Tools red141
red21. Wine Builtin DLLs red142

Chapter 14. Overview

Brief overview of Wine's architecture...

14.1. Basic Overview

Written by Ove Kaaven

With the fundamental architecture of Wine stabilizing, and people starting to think that we might soon be ready to actually release this thing, it may be time to take a look at how Wine actually works and operates.

14.1.1. Wine Overview

Wine is often used as a recursive acronym, standing for "Wine Is Not an Emulator". Sometimes it is also known to be used for "Windows Emulator". In a way, both meanings are correct, only seen from different perspectives. The first meaning says that Wine is not a virtual machine, it does not emulate a CPU, and you are not supposed to install neither Windows nor any Windows device drivers on top of it; rather, Wine is an implementation of the Windows API, and can be used as a library to port Windows applications to Unix. The second meaning, obviously, is that to Windows binaries (`.exe` files), Wine does look like Windows, and emulates its behaviour and quirks rather closely.

> **Note:** The "Emulator" perspective should not be thought of as if Wine is a typical inefficient emulation layer that means Wine can't be anything but slow - the faithfulness to the badly designed Windows API may of course impose a minor overhead in some cases, but this is both balanced out by the higher efficiency of the Unix platforms Wine runs on, and that other possible abstraction libraries (like Motif, GTK+, CORBA, etc) has a runtime overhead typically comparable to Wine's.

14.1.2. Win16 and Win32

Win16 and Win32 applications have different requirements; for example, Win16 apps expect cooperative multitasking among themselves, and to exist in the same address space, while Win32 apps except the complete opposite, i.e. preemptive multitasking, and separate address spaces.

Wine now deals with this issue by launching a separate Wine process for each Win32 process, but not for Win16 tasks. Win16 tasks are now run as different intersynchronized threads in the same Wine process; this Wine process is commonly known as a *WOW* process, referring to a similar mechanism used by Windows NT. Synchronization between the Win16 tasks running in the WOW process is normally done through the Win16 mutex - whenever one of them is running, it holds the Win16 mutex, keeping the others from running. When the task wishes to let the other tasks run, the thread releases the Win16 mutex, and one of the waiting threads will then acquire it and let its task run.

14.1.3. The Wineserver

The Wineserver is among the most confusing concepts in Wine. What is its function in Wine? Well, to be brief, it provides Inter-Process Communication (IPC), synchronization, and process/thread management. When the wineserver launches, it creates a Unix socket for the current host in your home directory's `.wine` subdirectory (or wherever the `WINEPREFIX` environment variable points) - all Wine processes launched later connects to the wineserver using this socket. (If a wineserver was not already running, the first Wine process will start up the wineserver in auto-terminate mode (i.e. the wineserver will then terminate itself once the last Wine process has terminated).)

Every thread in each Wine process has its own request buffer, which is shared with the wineserver. When a thread needs to synchronize or communicate with any other thread or process, it fills out its request buffer, then writes a command code through the socket. The wineserver handles the command as appropriate, while the client thread waits for a reply. In some cases, like with the various `WaitFor` synchronization primitives, the server handles it by marking the client thread as waiting and does not send it a reply before the wait condition has been satisfied.

The wineserver itself is a single and separate process and does not have its own threading - instead, it is built on top of a large `poll()` loop that alerts the wineserver whenever anything happens, such that a client has sent a command, or a wait condition has been satisfied. There is thus no danger of race conditions inside the wineserver itself - it is often called upon to do operations that look completely atomic to its clients.

Because the wineserver needs to manage processes, threads, shared handles, synchronization, and any related issues, all the client's Win32 objects are also managed by the wineserver, and the clients must send requests to the wineserver whenever they need to know any Win32 object handle's associated Unix file descriptor (in which case the wineserver duplicates the file descriptor, transmits it to the client, and leaves to the client to close the duplicate when it's done with it).

14.1.4. The Service Thread

The Wineserver cannot do everything that needs to be done behind the application's back, considering that it's not threaded (so cannot do anything that would block or take any significant amount of time), nor does it share the address space of its client threads. Thus, a special event loop also exists in each Win32 process' own address space, but handled like one of the process' own threads. This special thread is called the *service thread*, and does things that it wouldn't be appropriate for the wineserver to do. For example, it can call the application's asynchronous system timer callbacks every time a timer event is signalled (the wineserver handles the signalling, of course).
One important function of the service thread is to support the X11 driver's event loop. Whenever an event arrives from the X server, the service thread wakes up and sees the event, processes it, and posts messages into the application's message queues as appropriate. But this function is not unique - any number of Wine core components can install their own handlers into the service thread as necessary, whenever they need to do something independent of the application's own event loop. (At the moment, this includes, but is not limited to, multimedia timers, serial comms, and winsock async selects.)
The implementation of the service thread is in `scheduler/services.c`.

14.1.5. Relays, Thunks, and DLL descriptors

Loading a Windows binary into memory isn't that hard by itself, the hard part is all those various DLLs and entry points it imports and expects to be there and function as expected; this is, obviously, what the entire Wine implementation is all about. Wine contains a range of DLL implementations. Each of the implemented (or half-implemented) DLLs (which can be found in the `dlls/` directory) need to make themselves known to the Wine core through a DLL descriptor. These descriptors point to such things as the DLL's resources and the entry point table.
The DLL descriptor and entry point table is generated by the **winebuild** tool (previously just named **build**), taking DLL specification files with the extension `.spec` as input. The output file contains a global constructor that automatically registers the DLL's descriptor with the Wine core at runtime.
Once an application module wants to import a DLL, Wine will look through its list of registered DLLs (if it's not registered, it will look for it on disk). (Failing that, it will look for a real Windows `.DLL` file to use, and look through its imports, etc.) To resolve the module's imports, Wine looks through the entry point table and finds if it's defined there. (If not, it'll emit the error "No handler for ...", which, if the application called the entry point, is a fatal error.)
Since Wine is 32-bit code itself, and if the compiler supports Windows' calling convention, stdcall (**gcc** does), Wine can resolve imports into Win32 code by substituting the addresses of the Wine handlers directly without any thunking layer in between. This eliminates the overhead most people associate with "emulation", and is what the applications expect anyway.
However, if the user specified *`--debugmsg +relay`*, a thunk layer is inserted between the application imports and the Wine handlers; this layer is known as "relay" because all it does is print out the arguments/return values (by using the argument lists in the DLL descriptor's entry point table), then pass the call on, but it's invaluable for debugging misbehaving calls into Wine code. A similar mechanism also exists between Windows DLLs - Wine can optionally insert thunk layers between them, by using *`--debugmsg +snoop`*, but since no DLL descriptor information exists for non-Wine DLLs, this is less reliable and may lead to crashes.
For Win16 code, there is no way around thunking - Wine needs to relay between 16-bit and 32-bit code. These thunks switch between the app's 16-bit stack and Wine's 32-bit stack, copies and converts arguments as appropriate, and handles the Win16 mutex. Suffice to say that the kind of intricate stack content juggling this results in, is not exactly suitable study material for beginners.

14.1.6. Core and non-core DLLs

Wine must at least completely replace the "Big Three" DLLs (KERNEL/KERNEL32, GDI/GDI32, and USER/USER32), which all other DLLs are layered on top of. But since Wine is (for various reasons) leaning towards the NT way of implementing things, the NTDLL is another core DLL to be implemented in Wine, and many KERNEL32 and ADVAPI32 features will be implemented through the NTDLL. The wineserver and the service thread provide the backbone for the implementation of these core DLLs, and integration with the X11 driver (which provides GDI/GDI32 and USER/USER32 functionality along with the Windows standard controls).
All non-core DLLs, on the other hand, are expected to only use routines exported by other DLLs (and none of these backbone services directly), to keep the code base as tidy as possible. An example of this is COMCTL32 (Common Controls), which should only use standard GDI32- and USER32-exported routines.

14.2. Module Overview

written by (???)

(Extracted from `wine/documentation/internals`)

14.2.1. KERNEL Module

Needs some content...

14.2.2. GDI Module

14.2.2.1. X Windows System interface

The X libraries used to implement X clients (such as Wine) do not work properly if multiple threads access the same display concurrently. It is possible to compile the X libraries to perform their own synchronization (initiated by calling `XInitThreads()`). However, Wine does not use this approach. Instead Wine performs its own synchronization by putting a wrapper around every X call that is used. This wrapper protects library access with a critical section, and also arranges things so that X libraries compiled without `-D_REENTRANT` (eg. with global `errno` variable) will work with Wine.

To make this scheme work, all calls to X must use the proper wrapper functions (or do their own synchronization that is compatible with the wrappers). The wrapper for a function `X...()` is calles `TSX...()` (for "Thread Safe X ..."). So for example, instead of calling `XOpenDisplay()` in the code, `TSXOpenDisplay()` must be used. Likewise, X header files that contain function prototypes are wrapped, so that eg. `"ts_xutil.h"` must be included rather than `<X11/Xutil.h>`. It is important that this scheme is used everywhere to avoid the introduction of nondeterministic and hard-to-find errors in Wine.

The code for the thread safe X wrappers is contained in the `tsx11/` directory and in `include/ts*.h`. To use a new (ie. not previously used) X function in Wine, a new wrapper must be created. The wrappers are generated (semi-)automatically from the X11R6 includes using the `tools/make_X11wrappers` perl script. In simple cases it should be enough to add the name of the new function to the list in `tsx11/X11_calls`; if this does not work the wrapper must be added manually to the `make_X11wrappers` script. See comments in `tsx11/X11_calls` and `tools/make_X11wrappers` for further details.

14.2.3. USER Module

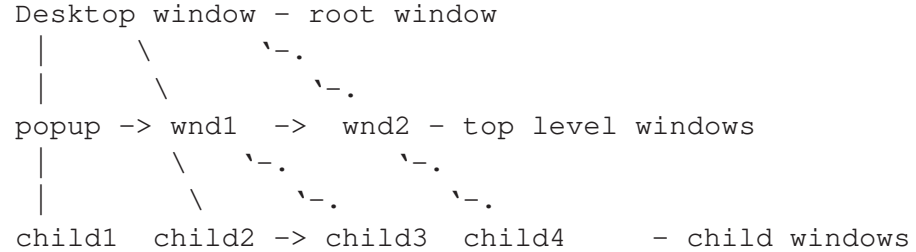
USER implements windowing and messaging subsystems. It also contains code for common controls and for other miscellaneous stuff (rectangles, clipboard, WNet, etc). Wine USER code is located in `windows/`, `controls/`, and `misc/` directories.

14.2.3.1. Windowing subsystem

`windows/win.c`

`windows/winpos.c`

Windows are arranged into parent/child hierarchy with one common ancestor for all windows (desktop window). Each window structure contains a pointer to the immediate ancestor (parent window if `WS_CHILD` style bit is set), a pointer to the sibling (returned by `GetWindow(..., GW_NEXT)`), a pointer to the owner window (set only for popup window if it was created with valid `hwndParent` parameter), and a pointer to the first child window (`GetWindow(.., GW_CHILD)`). All popup and non-child windows are therefore placed in the first level of this hierarchy and their ancestor link (`wnd->parent`) points to the desktop window.



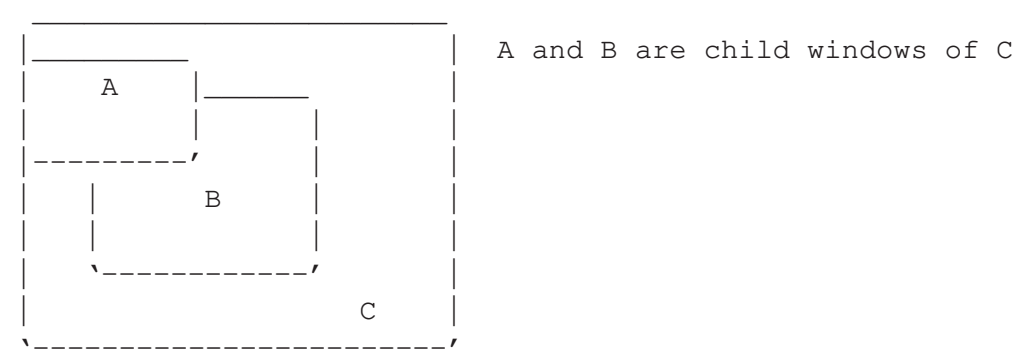
Horizontal arrows denote sibling relationship, vertical lines - ancestor/child. To summarize, all windows with the same immediate ancestor are sibling windows, all windows which do not have desktop as their immediate ancestor are child windows. Popup windows behave as topmost top-level windows unless they are owned. In this case the only requirement is that they must precede their owners in the top-level sibling list (they are not topmost). Child windows are confined to the client area of their parent windows (client area is where window gets to do its own drawing, non-client area consists of caption, menu, borders, intrinsic scrollbars, and minimize/maximize/close/help

Another fairly important concept is *z-order*. It is derived from the ancestor/child hierarchy and is used to determine "above/below" relationship. For instance, in the example above, z-order is `child1->popup->child2->child3->wnd1->child4->wnd2->desktop.`

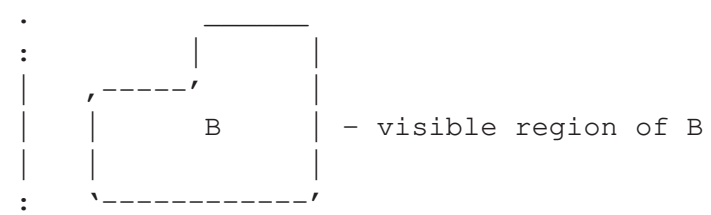
Current active window ("foreground window" in Win32) is moved to the front of z-order unless its top-level ancestor owns popup windows.
All these issues are dealt with (or supposed to be) in `windows/winpos.c` with `SetWindowPos()` being the primary interface to the window manager.
Wine specifics: in default and managed mode each top-level window gets its own X counterpart with desktop window being basically a fake stub. In desktop mode, however, only desktop window has an X window associated with it. Also, `SetWindowPos()` should eventually be implemented via `Begin/End/DeferWindowPos()` calls and not the other way around.

14.2.3.1.1. Visible region, clipping region and update region

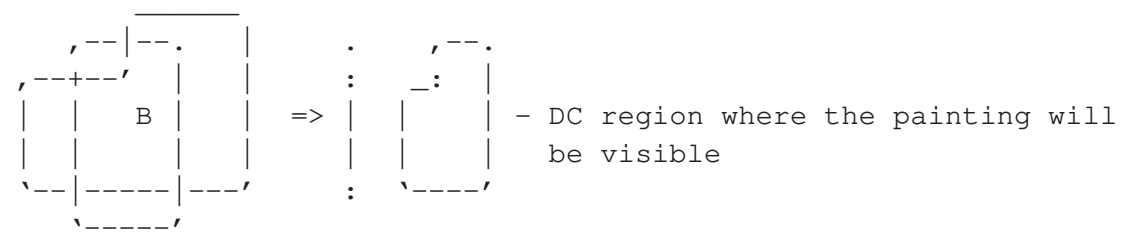
`windows/dce.c`
`windows/winpos.c`
`windows/painting.c`



Visible region determines which part of the window is not obscured by other windows. If a window has the `WS_CLIPCHILDREN` style then all areas below its children are considered invisible. Similarily, if the `WS_CLIPSIBLINGS` bit is in effect then all areas obscured by its siblings are invisible. Child windows are always clipped by the boundaries of their parent windows.
B has a `WS_CLIPSIBLINGS` style:

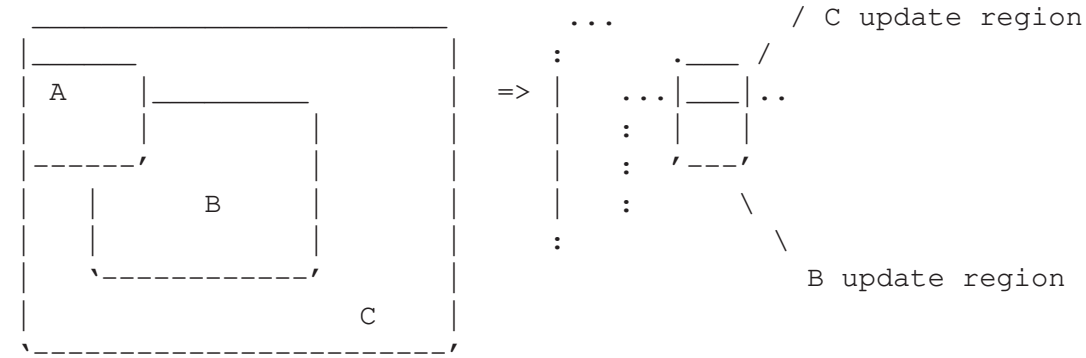


When the program requests a *display context* (DC) for a window it can specify an optional clipping region that further restricts the area where the graphics output can appear. This area is calculated as an intersection of the visible region and a clipping region.
Program asked for a DC with a clipping region:



When the window manager detects that some part of the window became visible it adds this area to the update region of this window and then generates `WM_ERASEBKGND` and `WM_PAINT` messages. In addition, `WM_NCPAINT` message is sent when the uncovered area intersects a nonclient part of the window. Application must reply to the `WM_PAINT` message by calling the `BeginPaint()`/`EndPaint()` pair of functions. `BeginPaint()` returns a DC that uses accumulated update region as a clipping region. This operation cleans up invalidated area and the window

A was moved to the left:



Windows maintains a display context cache consisting of entries that include the DC itself, the window to which it belongs, and an optional clipping region (visible region is stored in the DC itself). When an API call changes the state of the window tree, window manager has to go through the DC cache to recalculate visible regions for entries whose windows were involved in the operation. DC entries (DCE) can be either private to the window, or private to the window class, or shared between all windows (Windows 3.1 limits the number of shared DCEs to 5).

14.2.3.2. Messaging subsystem

`windows/queue.c`

`windows/message.c`

Each Windows task/thread has its own message queue - this is where it gets messages from. Messages can be:

1. generated on the fly (`WM_PAINT`, `WM_NCPAINT`, `WM_TIMER`)
2. created by the system (hardware messages)
3. posted by other tasks/threads (`PostMessage`)
4. sent by other tasks/threads (`SendMessage`)

Message priority:

First the system looks for sent messages, then for posted messages, then for hardware messages, then it checks if the queue has the "dirty window" bit set, and, finally, it checks for expired timers. See `windows/message.c`. From all these different types of messages, only posted messages go directly into the private message queue. System messages (even in Win95) are first collected in the system message queue and then they either sit there until `Get/PeekMessage` gets to process them or, as in Win95, if system queue is getting clobbered, a special thread ("raw input thread") assigns them to the private queues. Sent messages are queued separately and the sender sleeps until it gets a reply. Special messages are generated on the fly depending on the window/queue state. If the window update region is not empty, the system sets the `QS_PAINT` bit in the owning queue and eventually this window receives a `WM_PAINT` message (`WM_NCPAINT` too if the update region intersects with the non-client area). A timer event is raised when one of the queue timers expire. Depending on the timer parameters `DispatchMessage` either calls the callback function or the window procedure. If there are no messages pending the task/thread sleeps until messages appear.

There are several tricky moments (open for discussion) -

- System message order has to be honored and messages should be processed within correct task/thread context. Therefore when `Get/PeekMessage` encounters unassigned system message and this message appears not to be for the current task/thread it should either skip it (or get rid of it by moving it into the private message queue of the target task/thread - Win95, AFAIK) and look further or roll back and then yield until this message gets processed when system switches to the correct context (Win16). In the first case we lose correct message ordering, in the second case we have the infamous synchronous system message queue. Here is a post to one of the OS/2 newsgroup I found to be relevant:

  by David Charlap

  " Here's the problem in a nutshell, and there is no good solution. Every possible solution creates a different problem.

  With a windowing system, events can go to many different windows. Most are sent by applications or by the OS when things relating to that window happen (like repainting, timers, etc.)

  Mouse input events go to the window you click on (unless some window captures the mouse).

  So far, no problem. Whenever an event happens, you put a message on the target window's message queue. Every process has a message queue. If the process queue fills up, the messages back up onto the system queue.

  This is the first cause of apps hanging the GUI. If an app doesn't handle messages and they back up into the system queue, other apps can't get any more messages. The reason is that the next message in line can't go anywhere, and the

This can be fixed by making apps have bigger private message queues. The SIQ fix does this. PMQSIZE does this for systems without the SIQ fix. Applications can also request large queues on their own.

Another source of the problem, however, happens when you include keyboard events. When you press a key, there's no easy way to know what window the keystroke message should be delivered to.

Most windowing systems use a concept known as "focus". The window with focus gets all incoming keyboard messages. Focus can be changed from window to window by apps or by users clicking on winodws.

This is the second source of the problem. Suppose window A has focus. You click on window B and start typing before the window gets focus. Where should the keystrokes go? On the one hand, they should go to A until the focus actually changes to B. On the other hand, you probably want the keystrokes to go to B, since you clicked there first.

OS/2's solution is that when a focus-changing event happens (like clicking on a window), OS/2 holds all messages in the system queue until the focus change actually happens. This way, subsequent keystrokes go to the window you clicked on, even if it takes a while for that window to get focus.

The downside is that if the window takes a real long time to get focus (maybe it's not handling events, or maybe the window losing focus isn't handling events), everything backs up in the system queue and the system appears hung.

There are a few solutions to this problem.

One is to make focus policy asynchronous. That is, focus changing has absolutely nothing to do with the keyboard. If you click on a window and start typing before the focus actually changes, the keystrokes go to the first window until focus changes, then they go to the second. This is what X-windows does.

Another is what NT does. When focus changes, keyboard events are held in the system message queue, but other events are allowed through. This is "asynchronous" because the messages in the system queue are delivered to the application queues in a different order from that with which they were posted. If a bad app won't handle the "lose focus" message, it's of no consequence - the app receiving focus will get its "gain focus" message, and the keystrokes will go to it.

The NT solution also takes care of the application queue filling up problem. Since the system delivers messages asynchronously, messages waiting in the system queue will just sit there and the rest of the messages will be delivered to their apps.

The OS/2 SIQ solution is this: When a focus-changing event happens, in addition to blocking further messages from the application queues, a timer is started. When the timer goes off, if the focus change has not yet happened, the bad app has its focus taken away and all messages targetted at that window are skipped. When the bad app finally handles the focus change message, OS/2 will detect this and stop skipping its messages.

As for the pros and cons:

The X-windows solution is probably the easiest. The problem is that users generally don't like having to wait for the focus to change before they start typing. On many occasions, you can type and the characters end up in the wrong window because something (usually heavy system load) is preventing the focus change from happening in a timely manner.

The NT solution seems pretty nice, but making the system message queue asynchronous can cause similar problems to the X-windows problem. Since messages can be delivered out of order, programs must not assume that two messages posted in a particular order will be delivered in that same order. This can break legacy apps, but since Win32 always had an asynchronous queue, it is fair to simply tell app designers "don't do that". It's harder to tell app designers something like that on OS/2 - they'll complain "you changed the rules and our apps are breaking."

The OS/2 solution's problem is that nothing happens until you try to change window focus, and then wait for the timeout. Until then, the bad app is not detected and nothing is done."

- Intertask/interthread `SendMessage`. The system has to inform the target queue about the forthcoming message, then it has to carry out the context switch and wait until the result is available. Win16 stores necessary parameters in the queue structure and then calls `DirectedYield()` function. However, in Win32 there could be several messages pending sent by preemptively executing threads, and in this case `SendMessage` has to build some sort of message queue for sent messages. Another issue is what to do with messages sent to the sender when it is blocked inside its own `SendMessage`.

14.3. WINE/WINDOWS DLLs

Based upon various messages on wine-devel especially by Ulrich Weigand. Adapted by Michele Petrovski and Klaas van Gend.

(Extracted from `wine/documentation/dlls`)

This document mainly deals with the status of current DLL support by Wine. The Wine ini file currently supports settings to change the load order of DLLs. The load order depends on several issues, which results in different settings for various DLLs.

14.3.1. Pros of Native DLLs

Native DLLs of course guarantee 100% compatibility for routines they implement. For example, using the native

so on. Using the built-in WINE version of this library, on the other hand, would produce a display that does not precisely mimic that of Windows 95. Such subtle differences can be engendered in other important DLLs, such as the common controls library COMMCTRL or the common dialogs library COMMDLG, when built-in WINE DLLs outrank other types in load order.

More significant, less aesthetically-oriented problems can result if the built-in WINE version of the SHELL DLL is loaded before the native version of this library. SHELL contains routines such as those used by installer utilities to create desktop shortcuts. Some installers might fail when using WINE's built-in SHELL.

14.3.2. Cons of Native DLLs

Not every application performs better under native DLLs. If a library tries to access features of the rest of the system that are not fully implemented in Wine, the native DLL might work much worse than the corresponding built-in one, if at all. For example, the native Windows GDI library must be paired with a Windows display driver, which of course is not present under Intel Unix and WINE.

Finally, occassionally built-in WINE DLLs implement more features than the corresponding native Windows DLLs. Probably the most important example of such behavior is the integration of Wine with X provided by WINE's built-in USER DLL. Should the native Windows USER library take load-order precedence, such features as the ability to use the clipboard or drag-and- drop between Wine windows and X windows will be lost.

14.3.3. Deciding Between Native and Built-In DLLs

Clearly, there is no one rule-of-thumb regarding which load-order to use. So, you must become familiar with:

- what specific DLLs do
- which other DLLs or features a given library interacts with

and use this information to make a case-by-case decision.

14.3.4. Load Order for DLLs

Using the DLL sections from the wine configuration file, the load order can be tweaked to a high degree. In general it is advised not to change the settings of the configuration file. The default configuration specifies the right load order for the most important DLLs.

The default load order follows this algorithm: for all DLLs which have a fully-functional Wine implementation, or where the native DLL is known not to work, the built-in library will be loaded first. In all other cases, the native DLL takes load-order precedence.

The `DefaultLoadOrder` from the [DllDefaults] section specifies for all DLLs which version to try first. See manpage for explanation of the arguments.

The [DllOverrides] section deals with DLLs, which need a different-from-default treatment.

The [DllPairs] section is for DLLs, which must be loaded in pairs. In general, these are DLLs for either 16-bit or 32-bit applications. In most cases in Windows, the 32-bit version cannot be used without its 16-bit counterpart. For WINE, it is customary that the 16-bit implementations rely on the 32-bit implementations and cast the results back to 16-bit arguments. Changing anything in this section is bound to result in errors.

For the future, the Wine implementation of Windows DLL seems to head towards unifying the 16 and 32 bit DLLs wherever possible, resulting in larger DLLs. They are stored in the `dlls/` subdirectory using the 16-bit name. For large DLLs, a split might be discussed.

14.3.5. Understanding What DLLs Do

The following list briefly describes each of the DLLs commonly found in Windows whose load order may be modified during the configuration and compilation of WINE.

(See also `./DEVELOPER-HINTS` or the `dlls/` subdirectory to see which DLLs are currently being rewritten for wine)

```
ADVAPI32.DLL:    32-bit application advanced programming interfaces
                   like crypto, systeminfo, security and eventlogging
AVIFILE.DLL:    32-bit application programming interfaces for the
                   Audio Video Interleave (AVI) Windows-specific
                   Microsoft audio-video standard
COMMCTRL.DLL:    16-bit common controls
COMCTL32.DLL:    32-bit common controls
COMDLG32.DLL:    32-bit common dialogs
COMMDLG.DLL:    16-bit common dialogs
COMPOBJ.DLL:    OLE 16- and 32-bit compatibility libraries
```

```
CRTDLL.DLL:    Microsoft C runtime
DCIMAN.DLL:    16-bit
DCIMAN32.DLL:    32-bit display controls
DDEML.DLL:    DDE messaging
D3D*.DLL    DirectX/Direct3D drawing libraries
DDRAW.DLL:    DirectX drawing libraries
DINPUT.DLL:    DirectX input libraries
DISPLAY.DLL:    Display libraries
DPLAY.DLL, DPLAYX.DLL:  DirectX playback libraries
DSOUND.DLL:    DirectX audio libraries
GDI.DLL:    16-bit graphics driver interface
GDI32.DLL:    32-bit graphics driver interface
IMAGEHLP.DLL:    32-bit IMM API helper libraries (for PE-executables)
IMM32.DLL:    32-bit IMM API
IMGUTIL.DLL:
KERNEL32.DLL    32-bit kernel DLL
KEYBOARD.DLL:    Keyboard drivers
LZ32.DLL:    32-bit Lempel-Ziv or LZ file compression
                    used by the installshields (???).
LZEXPAND.DLL:    LZ file expansion; needed for Windows Setup
MMSYSTEM.DLL:    Core of the Windows multimedia system
MOUSE.DLL:    Mouse drivers
MPR.DLL:    32-bit Windows network interface
MSACM.DLL:    Core of the Addressed Call Mode or ACM system
MSACM32.DLL:    Core of the 32-bit ACM system
                   Audio Compression Manager ???
MSNET32.DLL    32-bit network APIs
MSVFW32.DLL:    32-bit Windows video system
MSVIDEO.DLL:    16-bit Windows video system
OLE2.DLL:    OLE 2.0 libraries
OLE32.DLL:    32-bit OLE 2.0 components
OLE2CONV.DLL:    Import filter for graphics files
OLE2DISP.DLL, OLE2NLS.DLL: OLE 2.1 16- and 32-bit interoperability
OLE2PROX.DLL:    Proxy server for OLE 2.0
OLE2THK.DLL:    Thunking for OLE 2.0
OLEAUT32.DLL    32-bit OLE 2.0 automation
OLECLI.DLL:    16-bit OLE client
OLECLI32.DLL:    32-bit OLE client
OLEDLG.DLL:    OLE 2.0 user interface support
OLESVR.DLL:    16-bit OLE server libraries
OLESVR32.DLL:    32-bit OLE server libraries
PSAPI.DLL:    Proces Status API libraries
RASAPI16.DLL:    16-bit Remote Access Services libraries
RASAPI32.DLL:    32-bit Remote Access Services libraries
SHELL.DLL:    16-bit Windows shell used by Setup
SHELL32.DLL:    32-bit Windows shell (COM object?)
TAPI/TAPI32/TAPIADDR:  Telephone API (for Modems)
W32SKRNL:    Win32s Kernel ? (not in use for Win95 and up!)
WIN32S16.DLL:    Application compatibility for Win32s
WIN87EM.DLL:    80387 math-emulation libraries
WINASPI.DLL:    Advanced SCSI Peripheral Interface or ASPI libraries
WINDEBUG.DLL    Windows debugger
WINMM.DLL:    Libraries for multimedia thunking
WING.DLL:    Libraries required to "draw" graphics
WINSOCK.DLL:    Sockets APIs
WINSPOOL.DLL:    Print spooler libraries
WNASPI32.DLL:    32-bit ASPI libraries
WSOCK32.DLL:    32-bit sockets APIs
```

Chapter 15. The Wine Debugger

15.1. I Introduction

written by Eric Pouech (Last updated: 8/15/2000)
(Extracted from `wine/documentation/winedbg`)

15.1.1. I.1 Processes and threads: in underlying OS and in Windows

Before going into the depths of debugging in Wine, here's a small overview of process and thread handling in Wine. It has to be clear that there are two different beasts: processes/threads from the Unix point of view and processes/threads from a Windows point of view.
Each Windows' thread is implemented as a Unix process (under Linux using the `clone` syscall), meaning that all threads of a same Windows' process share the same (unix) address space.
In the following:

- `W-process` means a process in Windows' terminology
- `U-process` means a process in Unix' terminology
- `W-thread` means a thread in Windows' terminology

A `W-process` is made of one or several `W-threads`. Each `W-thread` is mapped to one and only one `U-process`. All `U-processes` of a same `W-process` share the same address space.
Each Unix process can be identified by two values:

- the Unix process id (`upid` in the following)
- the Windows's thread id (`tid`)

Each Windows' process has also a Windows' process id (`wpid` in the following). It must be clear that `upid` and `wpid` are different and shall not be used instead of the other.
`Wpid` and `tid` are defined (Windows) system wide. They must not be confused with process or thread handles which, as any handle, is an indirection to a system object (in this case process or thread). A same process can have several different handles on the same kernel object. The handles can be defined as local (the values is only valid in a process), or system wide (the same handle can be used by any `W-process`).

15.1.2. I.2 Wine, debugging and WineDbg

When talking of debugging in Wine, there are at least two levels to think of:

- the Windows' debugging API.
- the Wine integrated debugger, dubbed **WineDbg**.

Wine implements most the the Windows' debugging API (the part in KERNEL32, not the one in `IMAGEHLP.DLL`), and allows any program (emulated or WineLib) using that API to debug a `W-process`.
WineDbg is a WineLib application making use of this API to allow debugging both any Wine or WineLib applications as well as Wine itself (kernel and all DLLs).
WineDbg understands symbolic information from both Unix world (mainly ELF stabs) and from Windows (most Microsoft debugging formats are supported - CodeView, .DBG files...)

15.2. II WineDbg's modes of invocation

15.2.1. II.1 Starting a process

Any application (either a Windows' native executable, or a WineLib application) can be run through **WineDbg**. Command line options and tricks are the same as for wine:

```
winedbg telnet.exe
winedbg "hl.exe -windowed"
```

15.2.2. II.2 Attaching

WineDbg can also be launched without any command line argument: **WineDbg** is started without any attached process. You can get a list of running `W-processes` (and their `wpid`'s) using the **walk process** command, and then, with the **attach** command, pick up the `wpid` of the `W-process` you want to debug. This is (for now) a neat feature for the following reasons:

- you can debug an already started application

15.2.3. II.3 On exception

When something goes wrong, Windows tracks this as an exception. Exceptions exist for segmentation violation, stack overflow, division by zero...

When an exception occurs, Wine checks if the `W-process` is debugged. If so, the exception event is sent to the debugger, which takes care of it: end of the story. This mechanism is part of the standard Windows' debugging API.

If the `W-process` is not debugged, Wine tries to launch a debugger. This debugger (normally **WineDbg**, see III Configuration for more details), at startup, attaches to the `W-process` which generated the exception event. In this case, you are able to look at the causes of the exception, and either fix the causes (and continue further the execution) or dig deeper to understand what went wrong.

If **WineDbg** is the standard debugger, the **pass** and **cont** commands are the two ways to let the process go further for the handling of the exception event.

To be more precise on the way Wine (and Windows) generates exception events, when a fault occurs (segmentation violation, stack overflow...), the event is first sent to the debugger (this is known as a first chance exception). The debugger can give two answers:

continue:
: the debugger had the ability to correct what's generated the exception, and is now able to continue process execution.

pass:
: the debugger couldn't correct the cause of the first chance exception. Wine will now try to walk the list of exception handlers to see if one of them can handle the exception. If no exception handler is found, the exception is sent once again to the debugger to indicate the failure of the exception handling.

  Note: since some of Wine's code uses exceptions and `try/catch` blocks to provide some functionality, **WineDbg** can be entered in such cases with segv exceptions. This happens, for example, with `IsBadReadPtr` function. In that case, the **pass** command shall be used, to let the handling of the exception to be done by the `catch` block in `IsBadReadPtr`.

15.2.4. II.4 Quitting

Unfortunately, Windows doesn't provide a detach kind of API, meaning that once you started debugging a process, you must do so until the process dies. Killing (or stopping/aborting) the debugger will also kill the debugged process. This will be true for any Windows' debugging API compliant debugger, starting with **WineDbg**.

15.3. III Configuration

15.3.1. III.1 Registry configuration

The Windows' debugging API uses a registry entry to know with debugger to invoke when an unhandled exception occurs (see II.3 for some details). Two values in key

```
"MACHINE\\Software\\Microsoft\\Windows NT\\CurrentVersion\\AeDebug"
```

Determine the behavior:

Debugger:
: this is the command line used to launch the debugger (it uses two `printf` formats (`%ld`) to pass context dependent information to the debugger). You should put here a complete path to your debugger (**WineDbg** can of course be used, but any other Windows' debugging API aware debugger will do).

Auto:
: if this value is zero, a message box will ask the user if he/she wishes to launch the debugger when an unhandled exception occurs. Otherwise, the debugger is automatically started.

A regular Wine registry looks like:

```
[MACHINE\\Software\\Microsoft\\Windows NT\\CurrentVersion\\AeDebug] 957636538
"Auto"=dword:00000001
"Debugger"="/usr/local/bin/winedbg %ld %ld"
```

Note 1: creating this key is mandatory. Not doing so will not fire the debugger when an exception occurs.

Note 2: wineinstall sets up this correctly. However, due to some limitation of the registry installed, if a previous Wine installation exists, it's safer to remove the whole

```
[MACHINE\\Software\\Microsoft\\Windows NT\\CurrentVersion\\AeDebug]
```

key before running again **wineinstall** to regenerate this key.

WineDbg can be configured thru a number of options. Those options are stored in the registry, on a per user basis. The key is (in *my* registry)

```
[eric\\Software\\Wine\\WineDbg]
```

Those options can be read/written while inside **WineDbg**, as part of the debugger expressions. To refer to one of this option, its name must be prefixed by a `$` sign. For example,

```
set $BreakAllThreadsStartup = 1
```

sets the option `BreakAllThreadsStartup` to `TRUE`.
All the options are read from the registry when **WineDbg** starts (if no corresponding value is found, a default value is used), and are written back to the registry when **WineDbg** exits (hence, all modifications to those options are automatically saved when **WineDbg** terminates).
Here's the list of all options:

15.3.2.1. III.2.1 Controling when the debugger is entered

`BreakAllThreadsStartup`
: Set to `TRUE` if at all threads start-up the debugger stops set to `FALSE` if only at the first thread startup of a given process the debugger stops. `FALSE` by default.

`BreakOnCritSectTimeOut`
: Set to `TRUE` if the debugger stops when a critical section times out (5 minutes); `TRUE` by default.

`BreakOnAttach`
: Set to `TRUE` if when **WineDbg** attaches to an existing process after an unhandled exception, **WineDbg** shall be entered on the first attach event. Since the attach event is meaningless in the context of an exception event (the next event which is the exception event is of course relevant), that option is likely to be `FALSE`.

`BreakOnFirstChance`
: An exception can generate two debug events. The first one is passed to the debugger (known as a first chance) just after the exception. The debugger can then decides either to resume execution (see **WineDbg**'s **cont** command) or pass the exception up to the exception handler chain in the program (if it exists) (**WineDbg** implements this thru the **pass** command). If none of the exception handlers takes care of the exception, the exception event is sent again to the debugger (known as last chance exception). You cannot pass on a last exception. When the `BreakOnFirstChance` exception is `TRUE`, then winedbg is entered for both first and last chance execptions (to `FALSE`, it's only entered for last chance exceptions).

15.3.2.2. III.2.2 Output handling

`ConChannelMask`
: Mask of active debugger output channels on console

`StdChannelMask`
: Mask of active debugger output channels on `stderr`

`UseXTerm`
: Set to `TRUE` if the debugger uses its own **xterm** window for console input/output. Set to `FALSE` if the debugger uses the current Unix console for input/output

Those last 3 variables are jointly used in two generic ways:

1. default

   ```
   ConChannelMask = DBG_CHN_MESG (1)
   StdChannelMask = 0
   UseXTerm = 1
   ```

   In this case, all input/output goes into a specific **xterm** window (but all debug messages `TRACE`, `WARN`... still goes to tty where wine is run from).
2. to have all input/output go into the tty where Wine was started from (to be used in a X11-free environment)

   ```
   ConChannelMask = 0
   StdChannelMask = DBG_CHN_MESG (1)
   UseXTerm = 1
   ```

Those variables also allow, for example for debugging purposes, to use:

```
ConChannelMask = 0xfff
StdChannelMask = 0xfff
```

```
UseXTerm = 1
```

This allows to redirect all `WineDbg` output to both tty Wine was started from, and **xterm** debugging window. If Wine (or **WineDbg**) was started with a redirection of `stdout` and/or `stderr` to a file (with for example >& shell redirect command), you'll get in that file both outputs. It may be interesting to look in the relay trace for specific values which the process segv'ed on.

15.3.2.3. III.2.2 Context information

`ThreadId`
: ID of the `W-thread` currently examined by the debugger

`ProcessId`
: ID of the `W-thread` currently examined by the debugger

<registers>
: All CPU registers are also available

The `ThreadId` and `ProcessId` variables can be handy to set conditional breakpoints on a given thread or process.

15.3.3. III.3 Finding files

WineDbg uses some lookup algorithms to find the files containing the debugging information. For ELF files, the current directory, the list of directories pointed by PATH, LD_LIBRARY_PATH are searched (in that order). For Microsoft debugging files, current directory, and directories pointed by _NT_SYMBOL_PATH and _NT_ALT_SYMBOL_PATH (in that order) are searched.

15.4. IV WineDbg commands

15.4.1. IV.1 Misc

```
abort aborts the debugger
quit exits the debugger

attach N attach to a W-process (N is its ID). IDs can be
obtained thru walk process command

help prints some help on the commands
help info prints some help on info commands

mode 16 switch to 16 bit mode
mode 32 switch to 32 bit mode
```

15.4.2. IV.2 Flow control

```
cont continue execution until next breakpoint or exception.
pass pass the exception event up to the filter chain.
step continue execution until next C line of code (enters
function call)
next  continue execution until next C line of code (doesn't
enter function call)
stepi execute next assembly instruction (enters function
call)
nexti execute next assembly instruction (doesn't enter
function call)
finish do nexti commands until current function is exited
```

cont, step, next, stepi, nexti can be postfixed by a number (N), meaning that the command must be executed N times.

15.4.3. IV.3 Breakpoints, watch points

```
enable N enables (break|watch)point #N
disable N disables (break|watch)point #N
delete N deletes  (break|watch)point #N
cond N removes any a existing condition to (break|watch)point N
cond N <expr> adds condition <expr> to (break|watch)point N. <expr>
```

```
will be evaluated each time the breakpoint is hit. If
the result is a zero value, the breakpoint isn't
triggered
break * N adds a breakpoint at address N
break <id> adds a breakpoint at the address of symbol <id>
break <id> N adds a breakpoint at the address of symbol <id> (N ?)
break N adds a breakpoint at line N of current source file
break adds a breakpoint at current $pc address
watch * N adds a watch command (on write) at address N (on 4 bytes)
watch <id> adds a watch command (on write) at the address of
symbol <id>
info break lists all (break|watch)points (with state)
```

15.4.4. IV.4 Stack manipulation

```
bt print calling stack of current thread
up goes up one frame in current thread's stack
up N goes up N frames in current thread's stack
dn goes down one frame in current thread's stack
dn N goes down N frames in current thread's stack
frame N set N as the current frame
info local prints information on local variables for current
function
```

15.4.5. IV.5 Directory & source file manipulation

```
show dir
dir <pathname>
dir
symbolfile <pathname>

list lists 10 source lines from current position
list -  lists 10 source lines before current position
list N lists 10 source lines from line N in current file
list <path>:N lists 10 source lines from line N in file <path>
list <id> lists 10 source lines of function <id>
list * N lists 10 source lines from address N
```

You can specify the end target (to change the 10 lines value) using the ',', For example:

```
list 123, 234  lists source lines from line 123 up to line 234 in
current file
list foo.c:1,56 lists source lines from line 1 up to 56 in file foo.c
```

15.4.6. IV.6 Displaying

A display is an expression that's evaluated and printed after the execution of any **WineDbg** command.

```
display lists the active displays
info display (same as above command)
display <expr> adds a display for expression <expr>
display /fmt <expr> adds a display for expression <expr>. Printing
evaluated <expr> is done using the given format (see
print command for more on formats)
del display N deletes display #N
undisplay N (same as del display)
```

15.4.7. IV.7 Disassembly

```
disas disassemble from current position
disas <expr> disassemble from address <expr>
disas <expr>,<expr>disassembles code between addresses specified by
the two <expr>
```

15.4.8. IV.8 Information on Wine's internals

```
info class <id> prints information on Windows's class <id>
walk class lists all Windows' class registered in Wine
info share lists all the dynamic libraries loaded the debugged
program (including .so files, NE and PE DLLs)
info module N prints information on module of handle N
walk module lists all modules loaded by debugged program
info queue N prints information on Wine's queue N
walk queue lists all queues allocated in Wine
info regs prints the value of CPU register
info segment N prints information on segment N
info segment lists all allocated segments
info stack prints the values on top of the stack
info map lists all virtual mappings used by the debugged
program
info wnd N prints information of Window of handle N
walk wnd lists all the window hierarchy starting from the
desktop window
walk wnd N lists all the window hierarchy starting from the
window of handle N
walk process lists all w-processes in Wine session
walk thread lists all w-threads in Wine session
walk modref (no longer avail)
```

15.4.9. IV.9 Memory (reading, writing, typing)

```
x <expr> examines memory at <expr> address
x /fmt <expr> examines memory at <expr> address using format /fmt
print <expr> prints the value of <expr> (possibly using its type)
print /fmt <expr> prints the value of <expr> (possibly using its
type)
set <lval>=<expr> writes the value of <expr> in <lval>
whatis <expr> prints the C type of expression <expr>
```

`/fmt` is either `/<letter>` or `/<count><letter>` letter can be

```
s => an ASCII string
u => an Unicode UTF16 string
i => instructions (disassemble)
x => 32 bit unsigned hexadecimal integer
d => 32 bit signed decimal integer
w => 16 bit unsigned hexadecimal integer
c => character (only printable 0x20-0x7f are actually
     printed)
b => 8 bit unsigned hexadecimal integer
```

15.5. V Other debuggers

15.5.1. V.1 Using other Unix debuggers

You can also use other debuggers (like **gdb**), but you must be aware of a few items:

You need to attach the unix debugger to the correct unix process (representing the correct windows thread) (you can "guess" it from a **ps fax** for example: When running the emulator, usually the first two `upids` are for the Windows' application running the desktop, the first thread of the application is generally the third `upid`; when running a WineLib program, the first thread of the application is generally the first `upid`)

> **Note:** Even if latest **gdb** implements the notion of threads, it won't work with Wine because the thread abstraction used for implementing Windows' thread is not 100% mapped onto the linux posix threads implementation. It means that you'll have to spawn a different **gdb** session for each Windows' thread you wish to debug.

15.5.2. V.2 Using other Windows debuggers

You can use any Windows' debugging API compliant debugger with Wine. Some reports have been made of success with VisualStudio debugger (in remote mode, only the hub runs in Wine). GoVest fully runs in Wine.

15.5.3. V.3 Main differences between winedbg and regular Unix debuggers

WineDbg	gdb
WineDbg debugs a Windows' process: + the various threads will be handled by the same WineDbg session + a breakpoint will be triggered for any thread of the w-process	gdb debugs a Windows' thread: + a separate gdb session is needed for each thread of Windows' process + a breakpoint will be triggered only for the w-thread debugged
WineDbg supports debug information from: + stabs (standard Unix format) + Microsoft's C, CodeView, .DBG	gdb supports debug information from: + stabs (standard Unix format)

15.6. VI Limitations

16 bit processes are not supported (but calls to 16 bit code in 32 bit applications are).

Chapter 16. Consoles in Wine

16.1. Consoles

written by (???)
(Extracted from `wine/documentation/console`)

16.1.1. Console - First Pass

Consoles are just xterms created with the *`-Sxxn`* switch. A pty is opened and the master goes to the `xterm` side and the slave is held by the wine side. The console itself it turned into a few HANDLE32s and is set to the `STD_*_HANDLES`.
It is possible to use the `WriteFile` and `ReadFile` commands to write to a win32 console. To accomplish this, all K32OBJs that support I/O have a read and write function pointer. So, `WriteFile` calls `K32OBJ_WriteFile` which calls the K32OBJ's write function pointer, which then finally calls `write`.
[this paragraph is now out of date] If the command line console is to be inheirited or a process inherits its parent's console (-- can that happen???), the console is created at process init time via `PROCESS_InheritConsole`. The `0`, `1`, and `2` file descriptors are duped to be the `STD_*_HANDLES` in this case. Also in this case a flag is set to indicate that the console comes from the parent process or command line.
If a process doesn't have a console at all, its `pdb->console` is set to `NULL`. This helps indicate when it is possible to create a new console (via `AllocConsole`).
When `FreeConsole` is called, all handles that the process has open to the console are closed. Like most K32OBJs, if the console's refcount reaches zero, its K32OBJ destroy function is called. The destroy kills the xterm if one was open.
Also like most k32 objects, we assume that (K32OBJ) header is the first field so the casting (from K32OBJ*to CONSOLE*) works correctly.
`FreeConsole` is called on process exit (in `ExitProcess`) if `pdb->console` is not `NULL`.

16.1.2. BUGS

Console processes do not inherit their parent's handles. I think there needs to be two cases, one where they have to inherit the `stdin` / `stdout` / `stderr` from unix, and one where they have to inherit from another windows app.
`SetConsoleMode` -- UNIX only has `ICANON` and various `ECHO`s to play around with for processing input. Win32 has line-at-a-time processing, character processing, and echo. I'm putting together an intermediate driver that will handle this (and hopefully won't be any more buggy then the NT4 console implementation).

16.1.3. Experimentation

experimentation with NT4 yields that:
`WriteFile`

- does not truncate file on 0 length write
- 0 length write or error on write changes `numcharswritten` to `0`
- 0 length write returns `TRUE`
- works with console handles

`_lwrite`

- does truncate/expand file at current position on 0 length write
- returns 0 on a zero length write
- works with console handles (typecasted)

`WriteConsole`

- expects only console handles

`SetFilePointer`

- returns -1 (err 6) when used with a console handle

`FreeConsole`

- even when all the handles to it are freed, the win32 console stays visible, the only way I could find to free it was via the `FreeConsole`

Is it possible to interrupt win32's `FileWrite`? I'm not sure. It may not be possible to interrupt any system calls.

16.1.4. DOS (Generic) Console Support

16.1.4.1. I. Command Line Configuration

DOS consoles must be configured either on the command line or in a dot resource file (`.console`). A typical configuration consists of a string of driver keywords separated by plus ('+') signs. To change the configuration on the command-line, use the *`-console`* switch.

For example:

```
wine -console ncurses+xterm <application>
```

Possible drivers:

tty:

Generic text-only support. Supports redirection.

ncurses:

Full-screen graphical support with color.

xterm:

Load a new window to display the console in. Also supports resizing windows.

16.1.4.2. II. `wine.conf` Configuration

In the `wine.conf` file, you can create a section called [console] that contains configuration options that are respected by the assorted console drivers.

Current Options:

XtermProg=<program>

Use this program instead of **xterm**. This eliminates the need for a recompile. See the table below for a comparison of various terminals.

InitialRows=<number>

Attempt to start all drivers with this number of rows. This causes xterms to be resized, for instance.

Note: This information is passed on the command-line with the *`-g`* switch.

InitialColumns=<number>

Attempt to start all drivers with this number of columns. This causes xterms to be resized, for instance.

Note: This information is passed on the command-line with the *`-g`* switch.

TerminalType=<name>

Tell any driver that is interested (ncurses) which termcap and/or terminfo type to use. The default is xterm which is appropiate for most uses. **nxterm** may give you better support if you use that terminal. This can also be changed to "linux" (or "console" on older systems) if you manage to hack the ability to write to the console into this driver.

16.1.4.3. III. Terminal Types

There are a large number of potential terminals that can be used with Wine, depending on what you are trying to do. Unfortunately, I am still looking for the "best" driver combination.

Note: 'slave' is required for use in Wine, currently.

Program	Color?	Resizing?	Slave?
xterm	N	Y	Y
nxterm	Y	N	Y
rxvt	Y	?	N
(linux console)	Y	N	?

As X terminals typically use a 24x80 screen resolution rather than the typical 25x80 one, it is necessary to resize the screen to allow a DOS program to work full-screen. There is a `wine.conf` option to work around this in some cases but run-time resizing will be disabled.

Chapter 17. Low-level Implementation

Details of Wine's Low-level Implementation...

17.1. Builtin DLLs

written by <juergen.schmied@metronet.de>

(Extracted from `wine/documentation/internal-dll`)

This document describes some points you should know before implementing the internal counterparts to external DLL's. Only 32 bit DLL's are considered.

17.1.1. 1. The LibMain function

This is the way to do some initializing when a process or thread is attached to the dll. The function name is taken from a `*.spec` file line:

```
init    YourFunctionName
```

Then, you have to implement the function:

```
BOOL32 WINAPI YourLibMain(HINSTANCE32 hinstDLL,
 DWORD fdwReason, LPVOID lpvReserved)
{ if (fdwReason==DLL_PROCESS_ATTACH)
  { ...
  }
  ....
}
```

17.1.2. 2. Using functions from other built-in DLL's

The problem here is, that you can't know if you have to call the function from the internal or the external DLL. If you just call the function you will get the internal implementation. If the external DLL is loaded the executed program will use the external DLL and you the internal one. When you -as an example- fill an iconlist placed in the internal DLL the application won't get the icons from the external DLL.

To work around this, you should always use a pointer to call such functions:

```
/* definition of the pointer type*/
void (CALLBACK* pDLLInitComctl)();

/* getting the function address  this should be done in the
 LibMain function when called with DLL_PROCESS_ATTACH*/

BOOL32 WINAPI Shell32LibMain(HINSTANCE32 hinstDLL, DWORD fdwReason,
 LPVOID lpvReserved)
{ HINSTANCE32 hComctl32;
  if (fdwReason==DLL_PROCESS_ATTACH)
  { /* load the external / internal DLL*/
    hComctl32 = LoadLibrary32A("COMCTL32.DLL");
    if (hComctl32)
    { /* get the function pointer */
      pDLLInitComctl=GetProcAddress32(hComctl32,"InitCommonControlsEx");

  /* check it */
      if (pDLLInitComctl)
      { /* use it */
        pDLLInitComctl();
      }

      /* free the DLL / decrease the ref count */
      FreeLibrary32(hComctl32);
    }
    else
    { /* do some panic*/
      ERR(shell,"P A N I C error getting functionpointers\n");
```

```
    }
  }
 ....
```

17.1.3. 3. Getting resources from a `*.rc` file linked to the DLL

< If you know how, write some lines>

17.2. Accelerators

Findings researched by Uwe Bonnes, Ulrich Weigand and Marcus Meissner.
(Extracted from `wine/documentation/accelerators`)
Some notes concerning accelerators.
There are *three* differently sized accelerator structures exposed to the user. The general layout is:

```
BYTE    fVirt;
WORD    key;
WORD    cmd;
```

We now have three different appearances:

1. Accelerators in NE resources. These have a size of 5 byte and do not have any padding. This is also the internal layout of the global handle HACCEL (16 and 32) in Windows 95 and WINE. Exposed to the user as Win16 global handles HACCEL16 and HACCEL32 by the Win16/Win32 API.
2. Accelerators in PE resources. These have a size of 8 byte. Layout is:

   ```
   BYTE    fVirt;
   BYTE    pad0;
   WORD    key;
   WORD    cmd;
   WORD    pad1;
   ```

   They are exposed to the user only by direct accessing PE resources.
3. Accelerators in the Win32 API. These have a size of 6 bytes. Layout is:

   ```
   BYTE    fVirt;
   BYTE    pad0;
   WORD    key;
   WORD    cmd;
   ```

   These are exposed to the user by the `CopyAcceleratorTable` and `CreateAcceleratorTable` functions in the Win32 API.

Why two types of accelerators in the Win32 API? We can only guess, but my best bet is that the Win32 resource compiler can/does not handle struct packing. Win32 ACCEL is defined using `#pragma(2)` for the compiler but without any packing for RC, so it will assume `#pragma(4)`.

17.3. File Handles

written by (???)
(Extracted from `wine/documentation/filehandles`)
DOS treats the first 5 file handles as special cases. They map directly to `stdin`, `stdout`, `stderr`, `stdaux` and `stdprn`. Windows 16 inherits this behavior, and in fact, win16 handles are interchangable with DOS handles. Some nasty windows programs even do this!
Windows32 issues file handles starting from `1`, on the grounds that most GUI processes don't need a `stdin`, `stdout`, etc.
The Wine handle code is implemented in the Win32 style, and the Win16 functions use two macros to convert to and from the two types.
The macros are defined in `file.h` as follows:

```
#define HFILE16_TO_HFILE32(handle) \
(((handle)==0) ? GetStdHandle(STD_INPUT_HANDLE) : \
 ((handle)==1) ? GetStdHandle(STD_OUTPUT_HANDLE) : \
 ((handle)==2) ? GetStdHandle(STD_ERROR_HANDLE) : \
 ((handle)>0x400) ? handle : \
```

```
#define HFILE32_TO_HFILE16(handle) ({ HFILE32 hnd=handle; \
      ((hnd==HFILE_ERROR32) ? HFILE_ERROR16 : \
      ((handle>0x400) ? handle : \
       (HFILE16)hnd+5); })
```

Warning

Be careful not to use the macro `HFILE16_TO_HFILE32` on functions with side-effects, as it will cause them to be evaluated several times. This could be considered a bug, but the use of this macro is limited enough not to need a rewrite.

Note: The `0x400` special case above deals with LZW filehandles (see `misc/lzexpand.c`).

17.4. Doing A Hardware Trace In Wine

written by Jonathan Buzzard <jab@hex.prestel.co.uk>

(Extracted from `wine/documentation/ioport-trace-hints`)

The primary reason to do this is to reverse engineer a hardware device for which you don't have documentation, but can get to work under Wine.

This lot is aimed at parallel port devices, and in particular parallel port scanners which are now so cheap they are virtually being given away. The problem is that few manufactures will release any programming information which prevents drivers being written for Sane, and the traditional technique of using DOSemu to produce the traces does not work as the scanners invariably only have drivers for Windows.

Please note that I have not been able to get my scanner working properly (a UMAX Astra 600P), but a couple of people have reported success with at least the Artec AS6e scanner. I am not in the process of developing any driver nor do I intend to, so don't bug me about it. My time is now spent writing programs to set things like battery save options under Linux on Toshiba laptops, and as such I don't have any spare time for writing a driver for a parallel port scanner etc.

Presuming that you have compiled and installed wine the first thing to do is is to enable direct hardware access to your parallel port. To do this edit `wine.conf` (usually in `/usr/local/etc`) and in the ports section add the following two lines

```
read=0x378,0x379,0x37a,0x37c,0x77a
write=0x378,x379,0x37a,0x37c,0x77a
```

This adds the necessary access required for SPP/PS2/EPP/ECP parallel port on LPT1. You will need to adjust these number accordingly if your parallel port is on LPT2 or LPT0.

When starting wine use the following command line, where `XXXX` is the program you need to run in order to access your scanner, and `YYYY` is the file your trace will be stored in:

```
wine -debugmsg +io XXXX 2> >(sed 's/^[^:]*:io:[^ ]* //' > YYYY)
```

You will need large amounts of hard disk space (read hundreds of megabytes if you do a full page scan), and for reasonable performance a really fast processor and lots of RAM.

You might well find the log compression program that `<David Campbell campbell@torque.net>` wrote helpful in reducing the size of the log files. This can be obtained by the following command:

```
sh ioport-trace-hints
```

This should extract `shrink.c` (which is located at the end of this file. Compile the log compression program by:

```
cc shrink.c -o shrink
```

Use the **shrink** program to reduce the physical size of the raw log as follows:

```
cat log | shrink > log2
```

The trace has the basic form of

```
XXXX > YY @ ZZZZ:ZZZZ
```

where `XXXX` is the port in hexidecimal being accessed, `YY` is the data written (or read) from the port, and `ZZZZ:ZZZZ` is the address in memory of the instruction that accessed the port. The direction of the arrow indicates whether the data was written or read from the port.

```
> data was written to the port
< data was read from the port
```

My basic tip for interperating these logs is to pay close attention to the addresses of the IO instructions. Their grouping and sometimes proximity should reveal the presence of subroutines in the driver. By studying the different versions you should be able to work them out. For example consider the following section of trace from my UMAX Astra 600P

```
0x378 > 55 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > aa @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
```

As you can see their is a repeating structure starting at address `0297:01ec` that consists of four io accesses on the parallel port. Looking at it the first io access writes a changing byte to the data port the second always writes the byte `0x05` to the control port, then a value which always seems to `0x8f` is read from the status port at which point a byte `0x04` is written to the control port. By studying this and other sections of the trace we can write a C routine that emulates this, shown below with some macros to make reading/writing on the parallel port easier to read.

```
#define r_dtr(x)        inb(x)
#define r_str(x)        inb(x+1)
#define r_ctr(x)        inb(x+2)
#define w_dtr(x,y)      outb(y, x)
#define w_str(x,y)      outb(y, x+1)
#define w_ctr(x,y)      outb(y, x+2)

/*
 * Seems to be sending a command byte to the scanner
 *
 */
int udpp_put(int udpp_base, unsigned char command)
{
        int loop,value;

        w_dtr(udpp_base, command);
        w_ctr(udpp_base, 0x05);

        for (loop=0;loop<10;loop++)
                if (((value=r_str(udpp_base)) & 0x80)!=0x00) {
                        w_ctr(udpp_base, 0x04);
```

```
                                }

        return (value & 0xf8) | 0x01;
}
```

For the UMAX Astra 600P only seven such routines exist (well 14 really, seven for SPP and seven for EPP). Whether you choose to disassemble the driver at this point to verify the routines is your own choice. If you do, the address from the trace should help in locating them in the disassembly.
You will probably then find it useful to write a script/perl/C program to analyse the logfile and decode them futher as this can reveal higher level grouping of the low level routines. For example from the logs from my UMAX Astra 600P when decoded futher reveal (this is a small snippet)

```
start:
put: 55 8f
put: aa 8f
put: 00 8f
put: 00 8f
put: 00 8f
put: c2 8f
wait: ff
get: af,87
wait: ff
get: af,87
end: cc
start:
put: 55 8f
put: aa 8f
put: 00 8f
put: 03 8f
put: 05 8f
put: 84 8f
wait: ff
```

From this it is easy to see that `put` routine is often grouped together in five successive calls sending information to the scanner. Once these are understood it should be possible to process the logs further to show the higher level routines in an easy to see format. Once the highest level format that you can derive from this process is understood, you then need to produce a series of scans varying only one parameter between them, so you can discover how to set the various parameters for the scanner.
The following is the `shrink.c` program.

```
cat > shrink.c <<EOF
#include <stdio.h>
#include <string.h>

void
main (void)
{
  char buff[256], lastline[256];
  int count;

  count = 0;
  lastline[0] = 0;

  while (!feof (stdin))
    {
      fgets (buff, sizeof (buff), stdin);
      if (strcmp (buff, lastline) == 0)
{
  count++;
}
```

```
{
  if (count > 1)
    fprintf (stdout, "# Last line repeated %i times #\n", count);
  fprintf (stdout, "%s", buff);
  strcpy (lastline, buff);
    count = 1;
}
    }
}
EOF
```

Chapter 18. Wine and OpenGL

written by Lionel Ulmer <lionel.ulmer@free.fr>, last modification : 2000/06/13
(Extracted from `wine/documentation/opengl`)

18.1. I What is needed to have OpenGL support in Wine

Basically, if you have a Linux OpenGL ABI compliant libGL (http://oss.sgi.com/projects/ogl-sample/ABI/ (http://oss.sgi.com/projects/ogl-sample/ABI/)) installed on your computer, you should everything that is needed.
To be more clear, I will detail one step after another what the **configure** script checks.
If, after Wine compiles, OpenGL support is not compiled in, you can always check `config.log` to see which of the following points failed.

18.1.1. I.1 Header files

The needed header files to build OpenGL support in Wine are :

`gl.h:`
: the definition of all OpenGL core functions, types and enumerants

`glx.h:`
: how OpenGL integrates in the X Window environment

`glext.h:`
: the list of all registered OpenGL extensions

The latter file (`glext.h`) is, as of now, not necessary to build Wine. But as this file can be easily obtained from SGI (http://oss.sgi.com/projects/ogl-sample/ABI/glext.h (http://oss.sgi.com/projects/ogl-sample/ABI/glext.h)), and that all OpenGL should provide one, I decided to keep it here.

18.1.2. I.2 OpenGL library thread-safety

After that, the script checks if the OpenGL library relies or not on the pthread library to provide thread safety (most 'modern' OpenGL libraries do).
If the OpenGL library explicitely links in libpthread (you can check it with a **ldd libGL.so**), you need to force OpenGL support by starting **configure** with the *`--enable-opengl`* flag.
The reason to this is that Wine contains some hacks done by Ove to cohabit with pthread that are known to work well in most of the cases (glibc 2.1.x). On the other hand, we never got Wine to work with glibc 2.0.6. Thus, I deemed preferable to play it safe : by default, I suppose that the hack won't work and that it's the user's responsability to enable it.
Anyway, it should be pretty safe to build with *`--enable-opengl`*.

18.1.3. I.3 OpenGL library itself

To check for the presence of 'libGL' on the system, the script checks if it defines the `glXCreateContext` function. There should be no problem here.

18.1.4. I.4 glXGetProcAddressARB function

The core of Wine's OpenGL implementation (at least for all extensions) is the `glXGetProcAddressARB` function. Your OpenGL library needs to have this function defined for Wine to be able to support OpenGL.
If your library does not provide it, you are out of luck.

> **Note:** this is not completely true as one could rewrite a `glXGetProcAddressARB` replacement using `dlopen` and friends, but well, telling people to upgrade is easier :-).

18.2. II How to configure

Configuration is quite easy : once OpenGL support has been built in Wine, this internal OpenGL driver will be used each time an application tries to load `opengl32.dll`.
Due to restrictions (that do not exist in Windows) on OpenGL contexts, if you want to prevent the screen to flicker when using OpenGL applications (all games are using double-buffered contexts), you need to set the following option in your `.winerc` / `wine.ini` in the [x11drv] section :

```
DesktopDoubleBuffered = Y
```

and to run Wine with the *`--desktop`* option.

18.3. III How it all works

The core OpenGL function calls are the same between Windows and Linux. So what is the difficulty to support it in Wine ? Well, there are two different problems :

1. the interface to the windowing system is different for each OS. It's called 'GLX' for Linux (well, for X Window) and 'wgl' for Windows. Thus, one need first to emulate one (wgl) with the other (GLX).
2. the calling convention between Windows (the 'Pascal' convention or 'stdcall') is different from the one used on Linux (the 'C' convention or 'cdecl'). This means that each call to an OpenGL function must be 'translated' and cannot be used directly by the Windows program.

Add to this some braindead programs (using GL calls without setting-up a context or deleting three time the same context) and you have still some work to do :-)

18.3.1. III.1 The Windowing system integration

This integration is done at two levels :

1. At GDI level for all pixel format selection routines (ie choosing if one wants a depth / alpha buffer, the size of these buffers, ...) and to do the 'page flipping' in double buffer mode. This is implemented in `graphics/x11drv/opengl.c` (all these functions are part of Wine's graphic driver function pointer table and thus could be reimplented if ever Wine works on another Windowing system than X).
2. In the `OpenGL32.DLL` itself for all other functionalities (context creation / deletion, querying of extension functions, ...). This is done in `dlls/opengl32/wgl.c`.

18.3.2. III.2 The thunks

The thunks are the Wine code that does the calling convention translation and they are auto-generated by a Perl script. In Wine's CVS tree, these thunks are already generated for you. Now, if you want to do it yourself, there is how it all works....

The script is located in `dlls/opengl32` and is called **make_opengl**. It requires Perl5 to work and takes two arguments :

1. The first is the path to the OpenGL registry. Now, you will all ask 'but what is the OpenGL registry ?' :-) Well, it's part of the OpenGL sample implementation source tree from SGI (more informations at this URL : http://oss.sgi.com/projects/ogl-sample/ (http://oss.sgi.com/projects/ogl-sample/).
   To summarize, these files contain human-readable but easily parsed information on ALL OpenGL core functions and ALL registered extensions (for example the prototype, the OpenGL version, ...).
2. the second is the OpenGL version to 'simulate'. This fixes the list of functions that the Windows application can link directly to without having to query them from the OpenGL driver. Windows is based, for now, on OpenGL 1.1, but the thunks that are in the CVS tree are generated for OpenGL 1.2.
   This option can have three values: `1.0`, `1.1` and `1.2`.

This script generates three files :

1. `opengl32.spec` gives Wine's linker the signature of all function in the `OpenGL32.DLL` library so that the application can link them. Only 'core' functions are listed here.
2. `opengl_norm.c` contains all the thunks for the 'core' functions. Your OpenGL library must provide ALL the function used in this file as these are not queried at run time.
3. `opengl_ext.c` contains all the functions that are not part of the 'core' functions. Contrary to the thunks in `opengl_norm.c`, these functions do not depend at all on what your libGL provides.
   In fact, before using one of these thunks, the Windows program first needs to 'query' the function pointer. At this point, the corresponding thunk is useless. But as we first query the same function in libGL and store the returned function pointer in the thunk, the latter becomes functional.

18.4. IV Known problems - shortcomings

18.4.1. IV.1 Missing GLU32.DLL

GLU is a library that is layered upon OpenGL. There is a 100% correspondence between the `libGLU.so` that is used on Linux and `GLU32.DLL`.

As for the moment, I did not create a set of thunks to support this library natively in Wine (it would easy to do, but I am waiting for a better solution than adding another autogenerated thunk file), you can always download anywhere on the net (it's free) a `GLU32.DLL` file (by browsing, for example, http://ftpsearch.lycos.com/ (http://ftpsearch.lycos.com/)).

18.4.2. IV.2 OpenGL not detected at configure time

18.4.3. IV.3 When running an OpenGL application, the screen flickers

See section (II) for how to create the context double-buffered and thus preventing this flicker effect.

18.4.4. IV.4 Wine gives me the following error message :

```
Extension defined in the OpenGL library but NOT in opengl_ext.c...
Please report (lionel.ulmer@free.fr) !
```

This means that the extension requested by the application is found in the libGL used by Linux (ie the call to `glXGetProcAddressARB` returns a non-`NULL` pointer) but that this string was NOT found in Wine's extension registry.

This can come from two causes :

1. The `opengl_ext.c` file is too old and needs to be generated again.
2. Use of obsolete extensions that are not supported anymore by SGI or of 'private' extensions that are not registered. An example of the former are `glMTexCoord2fSGIS` and `glSelectTextureSGIS` as used by Quake 2 (and apparently also by old versions of Half Life). If documentation can be found on these functions, they can be added to Wine's extension set.

If you have this, run with *`--debugmsg +opengl`* and send me <`lionel.ulmer@free.fr`> the TRACE.

18.4.5. IV.5 `libopengl32.so` is built but it is still not working

This may be caused by some missing functions required by `opengl_norm.c` but that your Linux OpenGL library does not provide.

To check for this, do the following steps :

1. create a dummy `.c` file :
   ```
   int main(void) {
       return 0;
   }
   ```

2. try to compile it by linking both libwine and libopengl32 (this command line supposes that you installed the Wine libraries in `/usr/local/lib`, YMMV) :
   ```
   gcc dummy.c -L/usr/local/lib -lwine -lopengl32
   ```

3. if it works, the problem is somewhere else (and you can send me an email). If not, you could re-generate the thunk files for OpenGL 1.1 for example (and send me your OpenGL version so that this problem can be detected at configure time).

Chapter 19. The Wine Build System

How the Wine build system works, and how to tweak it...

Chapter 20. Tools

20.1. bin2res

written by juergen.schmied@debitel.net (11/99)

(Extracted from `wine/documentation/resources`)

This document desribes tools for handling resources within wine

20.1.1. bin2res

This tool allows the editing of embedded binary resources within `*.rc` files. These resources are stored as hex dump so they can be stored within the cvs tree. This makes the editing of the embedded bitmaps and icons harder.

20.1.2. Create binary files from an `.rc` file

The resources in the `.rc` file have to be marked by a header:

```
/* BINRES idb_std_small.bmp */
IDB_STD_SMALL BITMAP LOADONCALL DISCARDABLE
{
  '42 4D 20 07 00 00 00 00 00 00 76 00 00 00 28 00'
```

`BINRES` is the keyword followed by a filename. **bin2res -d bin rsrc.rc** generates binary files from all marked resources. If the binary file is newer it gets not overwritten. To force overwriting use the *`-f`* switch.

20.1.3. Create a `.rc` file from binaries

Put a header followed by empty brackets in the `.rc` file.

```
/* BINRES idb_std_small.bmp */
{}
```

Then run **bin2res rsrc.rc**. It will merge the resources into the `.rc` file if the binary resources are newer than the.rc file. To force the resources into the `.rc` file use the *`-f`* switch. If there is already a resource with the same filename in the `.rc` file it gets overwritten.

20.1.4. output of bin2res

```
bash-2.03# ../../tools/bin2res -d bin shres.rc
[000.ico:c][003.ico:c][008.ico:s][015.ico:s][034.ico:s]
```

`s` means skipped, `c` means changed.

Chapter 21. Wine Builtin DLLs

A more detailed look at Wine's builtin DLLs...

21.1. Common Controls

Their development status and their UNDOCUMENTED features and functions

written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

(Extracted from `wine/documentation/common_controls`)

21.1.1. 1. Introduction

The information provided herein is based on the dll version 4.72 which is included in MS Internet Explorer 4.01.
All information about common controls should be collected in this document.
All Wine programmers are encouraged to add their knowledge to this document.

21.1.2. 2. General Information

Further information about common controls can be found in the MS Platform SDK and the MS Internet Client SDK (most recent). Information from these SDK's will NOT be repeated here. Only information which can NOT be found in these SDK's will be collected here. Some information in the SDK's mentioned above is (intentionally???) WRONG. Corrections to wrong information will be collected here too.

21.1.2.1. 2.1 Structure sizes of different common control versions

The common controls have been continously improved in the past. Some of the orignal structures had to be extended and their size changed. Most of the common control structures include their size as the first parameter. If a control gets the wrong size in a message or function a failure is very likely to occur. To avoid this, MS defined new constants that reflect the structure size of older `COMCTL32.DLL` versions. The following list shows the structure size constants that are currently defined in the original `COMCTL32.DLL`.

Note: Some stuctures are NOT defined in wine's COMCTL32 yet.

`HDITEM_V1_SIZE`:
: The size of the HDITEM structure in version 4.00.

`LVCOLUMN_V1_SIZE`:
: The size of the LVCOLUMN structure in version 4.00.

`LVHITTESTINFO_V1_SIZE`:
: The size of the LVHITTESTINFO structure in version 4.00.

`LVITEM_V1_SIZE`:
: The size of the LVITEM structure in version 4.00.

`NMLVCUSTOMDRAW_V3_SIZE`:
: The size of the NMLVCUSTOMDRAW structure in version 4.70.

`NMTTDISPINFO_V1_SIZE`:
: The size of the NMTTDISPINFO structure in version 4.00.

`NMTVCUSTOMDRAW_V3_SIZE`:
: The size of the NMTVCUSTOMDRAW structure in version 4.70.

`PROPSHEETHEADER_V1_SIZE`:
: The size of the PROPSHEETHEADER structure in version 4.00.

`PROPSHEETPAGE_V1_SIZE`:
: The size of the PROPSHEETPAGE structure in version 4.00.

`REBARBANDINFO_V3_SIZE`:
: The size of the REBARBANDINFO structure in version 4.70.

`TTTOOLINFO_V1_SIZE`:
: The size of the TOOLINFO structure in version 4.00.

`TVINSERTSTRUCT_V1_SIZE`:
: The size of the TVINSERTSTRUCT structure in version 4.00.

21.1.3. 3. Controls

This section describes the development status of the common controls.

21.1.3.1. 3.1 Animation Control

Author:

Dummy written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.2. 3.2 Combo Box Ex Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.3. 3.3 Date and Time Picker Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.4. 3.4 Drag List Box Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.5. 3.5 Flat Scroll Bar Control

Author:

Dummy written by Alex Priem. <alexp@sci.kun.nl>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.6. 3.6 Header Control

Author:

Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

- Almost finished.
- Unicode notifications are not supported (WM_NOTIFYFORMAT).
- Order array not supported.

21.1.3.7. 3.7 Hot Key Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.8. 3.8 Image List (no control)

Author:

Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Almost finished.

21.1.3.9. 3.9 IP Address Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>, Alex Priem <alexp@sci.kun.nl>

Status:

Under construction.

21.1.3.10. 3.10 List View Control

Author:

Dummy written by:

- Eric Kohl. <ekohl@abo.rhein-zeitung.de>
- Luc Tourangeau <luc@macadamian.com>
- Koen Deforche <jozef@kotnet.org>
- Francis Beaudet <francis@macadamian.com> and the "Corel-Team"

Status:

Under construction.

Notes:

Basic data structure with related messages are supported. No painting supported yet.

21.1.3.11. 3.11 Month Calendar Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.12. 3.12 Native font control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

21.1.3.13. 3.13 Pager Control

Author:

Dummy written by Eric Kohl. <ekohl@abo.rhein-zeitung.de>

Status:

Under construction. Many missing features.

Notes:

Author needed!! Any volunteers??

21.1.3.14. 3.14 Progress Bar Control

Author:

Original implementation by Dimitrie O. Paun. Fixes and improvements by Eric Kohl.

Status:

Finished!

21.1.3.15. 3.15 Property Sheet

Author:

Anders Carlsson <anders.carlsson@linux.nu> and Francis Beaudet <francis@macadamian.com>

Status:

Development in progress.

Notes:

Tab control must be implemented first.

21.1.3.16. 3.16 Rebar Control (Cool Bar)

Author:

Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Development in progress. Many bugs and missing features.

Notes:

Author needed!! Any volunteers??

21.1.3.17. 3.17 Status Bar Control

Author:

Original implementation by Bruce Milner. Fixes and improvements by Eric Kohl.

Status:

Almost finished.

Notes:

Tooltip integration is almost complete.

21.1.3.18. 3.18 Tab Control

Author:

Anders Carlsson <anders.carlsson@linux.nu>

Status:

Development in progress.

21.1.3.19. 3.19 Toolbar Control

Author:

Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Development in progress. Basic functionality is almost done. (dll version 4.0)

21.1.3.20. 3.20 Tooltip Control

Author:

Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Almost finished.

Notes:

Unicode support is incomplete (`WM_NOTIFYFORMAT`).

21.1.3.21. 3.21 Trackbar Control

Author:

Dummy written by Eric Kohl <ekohl@abo.rhein-zeitung.de>, Alex Priem <alexp@sci.kun.nl>

Status:

Under construction.

21.1.3.22. 3.22 Tree View Control

Author:

Dummy written by Eric Kohl., Alex Priem <alexp@sci.kun.nl>

Status:

Under construction.

21.1.3.23. 3.23 Updown Control

Author:

Original implementation by Dimitrie O. Paun. Some minor changes by Eric Kohl <ekohl@abo.rhein-zeitung.de>.

Status:

Unknown.

Notes: Have a look at `controls/updown.c` for a list of bugs and missing features.
The status is unknown, because I did not have a close look at this control. One test-program looked quite good, but in Win95's `cdplayer.exe` the control does not show at all.
Any volunteers??

21.1.4. 4. Additional Information

Has to be written...

21.1.5. 5. Undocumented features

There are quite a lot of undocumented functions like:

- DSA (Dynamic Storage Array) functions.
- DPA (Dynamic Pointer Array) functions.
- MRU ("Most Recently Used" List) functions.
- other unknown functions.

Have a look at `relay32/comctl32.spec`.

21.1.5.1. 5.1 Dymnamic Storage Array (DSA)

The DSA functions are used to store and manage dynamic arrays of fixed size memory blocks. They are used by `TASKMAN.EXE`, Explorer, IE4 and other Programs and DLL's that are "parts of the Windows Operating System".
The implementation should be complete.
Have a look at the source code to get more information.

21.1.5.2. 5.2 Dynamic Pointer Array (DPA)

Similar to the DSA functions, but they just store pointers. They are used by Explorer, IE4 and other Programs and DLL's that are "parts of the Windows Operating System". The implementation should be complete.
Have a look at the source code to get more information.

21.1.5.3. 5.3 "Most Recently Used" - List (MRU)

Only stubs are implemented to keep Explorer from bailing out.
No more information available at this time!

21.1.5.4. 5.4 MenuHelp

Has to be written...

21.1.5.5. 5.5 GetEffectiveClientRect

Has to be written...

21.1.5.6. 5.6 ShowHideMenuCtl

The official documentation provided by MS is incomplete.
`lpInfo`:

> Both values of the first pair must be the handle to the applications main menu.

21.1.5.7. 5.7 Other undocumented functions

Several other undocumented functions are used by IE4.
String functions: (will be written...)

21.1.6. 6. Epilogue

You see, much work has still to be done. If you are interested in writing a control send me an e-mail. If you like to fix bugs or add some functionality send an e-mail to the author of the control.