Wine Documentation

Wine Documentation

Wine User Guide

Wine User Guide

Table of Contents

Chapter 1. Introduction

What is Wine?

Written by John R. Sheets <jsheets@codeweavers.com>

Windows and Linux

Many people have faced the frustration of owning software that won't run on their computer. With the recent popularity of Linux, this is happening more and more often because of differing operating systems. Your Windows software won't run on Linux, and your Linux software won't run in Windows.

A common solution to this problem is to install both operating systems on the same computer, as a "dual boot" system. If you want to write a document in MS Word, you can boot up in Windows; if you want to run the GnuCash, the GNOME financial application, you can shut down your Windows session and reboot into Linux. The problem with this is that you can't do both at the same time. Each time you switch back and forth between MS Word and GnuCash, you have to reboot again. This can get tiresome quickly.

Life would be so much easier if you could run all your applications on the same system, regardless of whether they are written for Windows or for Linux. On Windows, this isn't really possible. [1] However, Wine makes it possible to run native Windows applications alongside native Linux applications on a Linux (or Solaris) system. You can share desktop space between MS Word and GnuCash, overlapping their windows, iconizing them, and even running them from the same launcher.

Emulation versus Native Linking

Wine is a UNIX implementation of the win32 libraries, written from scratch by hundreds of volunteer developers and released under an open source license. Anyone can download and read through the source code, and fix bugs that arise. The Wine community is full of richly talented programmers who have spent thousands of hours of personal time on improving Wine so that it works well with the win32 *Applications Programming Interface* (API), and keeps pace with new developments from Microsoft.

Wine can run applications in two discrete ways: as pre-compiled Windows binaries, or as natively compiled X11 (X Window System) applications. The former method uses emulation to connect a Windows application to the Wine libraries. You can run your Windows application directly with the emulator, by installing through Wine or by simply copying the Windows executables onto your Linux system.

The other way to run Windows applications with Wine requires that you have the source code for the application. Instead of compiling it with native Windows compilers, you can compile it with a native Linux compiler -- **gcc** for example -- and link in the Wine Libraries as you would with any other native UNIX application. These natively linked applications are referred to as Winelib applications.

The Wine Users Guide will focus on running precompiled Windows applications using the Wine emulator. The Winelib Users Guide will cover Winelib applications.

Wine Requirements and Features

Written by Andreas Mohr <amohr@codeweavers.com>

System requirements

In order to run Wine, you need the following:

- a computer ;-) Wine: only PCs >= i386 are supported at the moment. Winelib: other platforms might be supported, but can be tricky.
- a UNIX-like operating system such as Linux, *BSD, Solaris x86
- >= 16MB of RAM. Everything below is pretty much unusable. >= 64 MB is needed for a "good" execution.
- an X11 window system (XFree86 etc.). Wine is prepared for other graphics display drivers, but writing support is not too easy. The text console display driver is nearly usable.

Wine capabilities

Now that you hopefully managed to fulfill the requirements mentioned above, we tell you what Wine is able to do/support:

- Support for executing DOS, Win 3.x and Win9x/NT/Win2000 programs (most of Win32's controls are supported)
- Optional use of external vendor DLLs (e.g. original Windows DLLs)
- X11-based graphics display (remote display to any X terminal possible), text mode console
- Desktop-in-a-box or mixable windows
- Pretty advanced DirectX support for games
- Good support for sound, alternative input devices
- Printing: supports native Win16 printer drivers, Internal PostScript driver
- Modems, serial devices are supported
- Winsock TCP/IP networking
- ASPI interface (SCSI) support for scanners, CD writers, ...
- Unicode support, relatively advanced language support
- Wine debugger and configurable trace logging messages

Notes

1. Technically, if you have two networked computers, one running Windows and the other running Linux, and if you have some sort of X server software running on the Windows system, you can export Linux applications onto the Windows system. A free X server is available at http://xfree86.cygwin.com/. However, this doesn't solve the problem if you only own one computer system.

Chapter 2. Getting Wine

The Many Forms of Wine

The standard Wine distribution includes quite a few different executables, libraries, and configuration files. All of these must be set up properly for Wine to work well. This chapter will guide you through the necessary steps to get Wine installed on your system.

If you are running a distribution of Linux that uses packages to keep track of installed software, you may be in luck: A prepackaged version of Wine may already exist for your system. The first three sections will tell you how to find the latest Wine packages and get them installed. You should be careful, though, about mixing packages between different distributions, and even from different versions of the same distribution. Often a package will only work on the distribution it's compiled for. We'll cover Debian, Redhat, and other distributions.

If you're not lucky enough to have an available package for your operating system, or if you'd prefer a newer version of Wine than already exists as a package, you may have to download the Wine source code and compile it yourself on your own machine. Don't worry, it's not too hard to do this, especially with the many helpful tools that come with Wine. You don't need any programming experience to compile and install Wine, although it might be nice to have some minor UNIX administrative skill. We'll cover how to retrieve and compile the official source releases from the FTP archives, and also how to get the cutting edge up-to-the-minute fresh Wine source code from CVS (Concurrent Versions System). Both processes of source code installation are similar, and once you master one, you should have no trouble dealing with the other one.

Finally, you may someday need to know how to apply a source code patch to your version of Wine. Perhaps you've uncovered a bug in Wine, reported it to the Wine mailing list (mailto:wine-devel@winehq.com), and received a patch from a developer to hopefully fix the bug. The last section in this chapter will show you how to safely apply the patch and revert it if the patch doesn't work.

Getting Wine for a Debian System

In most cases on a Debian system, you can install Wine with a single command, as root:

```
# apt-get install wine
```

apt-get will connect to a Debian archive across the Internet (thus, you must be online), then download the Wine package and install it on your system. End of story.

Of course, Debian's pre-packaged version of Wine may not be the most recent release. If you are running the stable version of Debian, you may be able to get a slightly newer version of Wine by grabbing the package from the unstable distribution, although this may be a little risky, depending on how far the unstable distribution has diverged from the stable one. You can find a list of Wine binary packages for the various Debian releases using the package search engine at www.debian.org (http://www.debian.org).

To install a package that's not part of your distribution, you must use **dpkg** instead of **apt-get**. Since **dpkg** doesn't download the file for you, you must do it yourself. Follow the link on the package search engine to the desired package, then click on the Go To Download Page button and follow the instructions. Save the file to your hard drive, then run **dpkg** on it. For example, if you saved the file to your home directory, you might perform the following actions to install it:

```
$ su -
<Type in root password>
# cd /home/user
# dpkg -i wine_0.0.20000109-3.deb
```

You may also want to install the wine-doc package, and if you are using Wine from the 2.3 distribution (Woody), the wine-utils package as well.

Getting Wine for a Redhat System

Redhat/RPM users can use rpmfind.net (http://rpmfind.net/linux/RPM/) to track down available Wine RPM binaries. This page (http://rpmfind.net/linux/RPM/WByName.html) contains a list of all rpmfind packages that start with the letter "W", including a few Wine packages

Getting Wine for Other Distributions

The first place you should look if your system isn't Debian or Redhat is the WineHQ Download Page (http://www.winehq.com/download/). This page lists many assorted archives of binary (precompiled) Wine files.

Lycos FTPSearch (http://ftpsearch.lycos.com/?form=medium) is another useful resource for tracking down miscellaneous distribution packages.

Getting Wine Source Code from the FTP Archive

If the version of Wine you want does not exist in package form, you can download the source code yourself and compile it on your machine. Although this might seem a little intimidating at first if you've never done it, you'll find that it'll often go quite smoothly, especially on the newer Linux distributions.

The safest way to grab the source is from one of the official FTP archives. An up to date listing is in the ANNOUNCE (http://www.winehq.com/source/ANNOUNCE) file in the Wine distribution (which you would have if you already downloaded it). Here is a (possibly out of date) list of FTP servers carrying Wine:

- ftp://metalab.unc.edu/pub/Linux/ALPHA/wine/development/ (ftp://metalab.unc.edu/pub/Linux/ALPHA/wine/development/)
- ftp://tsx-11.mit.edu/pub/linux/ALPHA/Wine/development/ (ftp://tsx-11.mit.edu/pub/linux/ALPHA/Wine/development/)
- ftp://ftp.infomagic.com/pub/mirrors/linux/sunsite/ALPHA/wine/development/ (ftp://ftp.infomagic.com/pub/mirrors/linux/sunsite/ALPHA/wine/development/)
- ftp://orcus.progsoc.uts.edu.au/pub/Wine/development/ (ftp://orcus.progsoc.uts.edu.au/pub/Wine/development/)

The official releases are tagged by date with the format "Wine-YYYYMMDD.tar.gz". Your best bet is to grab the latest one.

FIXME: Explain how to un-tar, compile, and install Wine from a tarball.

Getting Wine Source Code from CVS

The official web page for Wine CVS is http://www.winehq.com/development/ (http://www.winehq.com/development/).

First, you need to get a copy of the latest Wine sources using CVS. You can tell it where to find the source tree by setting the CVSROOT environment variable. You also have to log in anonymously to the wine CVS server. In **bash**, it might look something like this:

```
$ export CVSROOT=:pserver:cvs@cvs.winehq.com:/home/wine
$ cvs login
Password: cvs
$ cvs checkout wine
```

That'll pull down the entire Wine source tree from winehq.com and place it in the current directory (actually in the 'wine' subdirectory). CVS has a million command line parameters, so there are many ways to pull down files, from anywhere in the revision history. Later, you can grab just the updates:

```
$ cvs -dP update
```

cvs update works from inside the source tree. You don't need the CVSROOT environment variable to run it either. You just have to be inside the source tree. The *-d* and *-P* options make sure your local Wine tree directory structure stays in sync with the remote repository.

After you've made changes, you can create a patch with **cvs diff -u**, which sends output to stdout (the *-u* controls the format of the patch). So, to create an `my_patch.diff` file, you would do this:

```
$ cvs diff -u > my_patch.diff
```

You can call **cvs diff** from anywhere in the tree (just like **cvs update**), and it will always grab recursively from that point. You can also specify single files or subdirectories:

```
$ cvs diff -u dlls/winaspi > my_aspi_patch.diff
```

Experiment around a little. It's fairly intuitive.

Upgrading Wine with a Patch

If you have the Wine source code, as opposed to a binary distribution, you have the option of applying patches to the source tree to fix bugs and add experimental features. Perhaps you've found a bug, reported it to the Wine mailing list (mailto:wine-devel@winehq.com), and received a patch file to fix the bug. You can apply the patch with the **patch** command, which takes a streamed patch from `stdin`:

```
$ cd wine
$ patch -p0 < ../patch_to_apply.diff
```

To remove the patch, use the *-R* option:

```
$ patch -p0 -R < ../patch_to_apply.diff
```

If you want to do a test run to see if the patch will apply successfully (e.g., if the patch was created from an older or newer version of the tree), you can use the *--dry-run* parameter to run the patch without writing to any files:

```
$ patch -p0 --dry-run < ../patch_to_apply.diff
```

patch is pretty smart about extracting patches from the middle of a file, so if you save an email with an inlined patch to a file on your hard drive, you can invoke patch on it without stripping out the email headers and other text. **patch** ignores everything that doesn't look like a patch.

FIXME: Go into more depth about the -p0 option...

Chapter 3. Installing/compiling Wine

How to install Wine...

WWN #52 Feature: Replacing Windows

Written by Ove Kåven <ovek@winehq.com>

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 source 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.

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.

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/config`:

```
[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.

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/.

Installing Wine Without Windows

Written by James Juran <`juran@cse.psu.edu`>

(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 DLLs 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/config` 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 DLLs 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 DLLs 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/config` to specify "native" before "builtin" for the Windows DLLs you want to use. For more information about this, see the Wine manpage.
- Note that some network DLLs 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 DLLs are "clean" to use). Make sure you have these specified in the "[DllPairs]" section of `~/.wine/config`.
- Be consistent: Use only DLLs from the same Windows version together.
- Put `regedit.exe` in the `C:\windows` directory. (Office 95 imports a `*.reg` file when it runs with an empty registry, don't know about Office 97).
- Also add `winhelp.exe` and `winhlp32.exe` if you want to be able to browse through your programs' help function.

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.

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/config` file. The following excerpt from a `~/.wine/config` 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 they 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 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 are 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.

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.

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`).

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 continues 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.

SCSI Support

Written by Bruce Milner <>; Additions by Andreas Mohr <`amohr@codeweavers.com`>

(Extracted from `wine/documentation/aspi`)

This file describes setting up the Windows ASPI interface.

> **Warning/Warning/Warning!!!!!!**
>
> This may trash your system if used incorrectly. It may even trash your system when 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) (well, for other devices such as CD burners, MO drives, ..., too)

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.

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. If you intend to use your IDE device, you need to use the ide-scsi emulation. Read http://www.linuxdoc.org/HOWTO/CD-Writing-HOWTO.html (http://www.linuxdoc.org/HOWTO/CD-Writing-HOWTO.html) for ide-scsi setup instructions.
2. Compile generic SCSI drivers into your kernel.
3. This seems to be not required any more for newer (2.2.x) kernels: 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 HOWTO at http://www.linuxdoc.org/HOWTO/SCSI-Programming-HOWTO.html (http://www.linuxdoc.org/HOWTO/SCSI-Programming-HOWTO.html) 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. For Win32 software (WNASPI32), Wine has auto-detection in place. For Win16 software (WINASPI), you need to add a SCSI device entry for your particular scanner to ~/.wine/config. 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.

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
```

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 4. Configuring Wine

Setting up config files, etc.

General Configuration

Copyright 1999 Adam Sacarny <magicbox@bestweb.net>

(Extracted from `wine/documentation/config`)

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
- Wine's Multimedia drivers and DLL configuration

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>/documentation/samples/config`. It is organized by section.

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
[x11drv]	recmd	Graphic driver settings
[fonts]	yes	Font appearance and recognition
[serialports]	no	COM ports seen by wine
[parallelports]	no	LPT ports seen by wine
[ppdev]	no	Parallelport emulation
[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

Section Name	Needed?	What it Does
[Console]	no	Console settings
[Clipboard]	no	Interaction for wine and X11 clipboard
[afmdirs]	no	Postscript driver settings
[WinMM]	yes	Multimedia settings
[AppDefaults]	no	Overwrite the settings of previous sections for special programs

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"
```

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. (The |'s mean "Type = '<one of the options>'".)

```
"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"
```

The [wine] Section

The [wine] section of the configuration file contains all kinds of general settings for Wine. When specifying the directories for the directory related 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"`).

```
"GraphicsDriver" = "x11drv|ttydrv"
```

Sets the graphics driver to use for Wine output. x11drv is for X11 output, ttydrv is for text console output. WARNING: if you use ttydrv here, then you won't be able to run any Windows GUI programs. Thus this option is mainly interesting for e.g. embedded use of Wine in web server scripts.

```
"Printer" = "off|on"
```

Tells wine whether to allow printing via printer drivers to work. This option isn't needed for our builtin psdrv printer driver at all. 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 via windows printer drivers, don't even add this to your wine config file (It probably isn't already in it). Check out the [spooler] and [parallelports] sections too.

```
"ShellLinker" = "wineshelllink"
```

This setting specifies the shell linker script to use for setting up Windows icons in e.g. KDE or Gnome that are given by programs making use of appropriate shell32.dll functionality to create icons on the desktop/start menu during installation.

```
"ShowDirSymlinks" = "1"
```

Wine doesn't pass directory symlinks to Windows programs by default, as doing so may crash some programs that do recursive lookups of whole subdirectory trees whenever a directory symlink points back to itself or one of its parent directories. That's why we disallowed the use of directory symlinks and added this setting to reenable ("1") this functionality.

```
"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.

Introduction To DLL Sections

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

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:

Win16	**Win32**	**Native** [a]
KERNEL	KERNEL32	No!
USER	USER32	No!
SHELL	SHELL32	Yes
GDI	GDI32	No!
COMMDLG	COMDLG32	Yes
VER	VERSION	Yes
Notes: a. Is it possible to use native dll with wine? (See next section)		

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.

The [DllDefaults] Section

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

```
"DefaultLoadOrder" =" native, so, builtin"
```

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

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 `~/.wine/config`, you may safely delete it.

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 config that was included with your wine package):

```
[DllOverrides]
"commdlg"      = "builtin, native"
"comdlg32"     = "builtin, native"
"ver"          = "builtin, native"
"version"      = "builtin, native"
"shell"        = "builtin, native"
"shell32"      = "builtin, native"
"lzexpand"     = "builtin, native"
"lz32"         = "builtin, native"
"comctl32"     = "builtin, native"
"commctrl"     = "builtin, native"
"wsock32"      = "builtin"
"winsock"      = "builtin"
"advapi32"     = "builtin, native"
"crtdll"       = "builtin, native"
"mpr"          = "builtin, native"
"winspool.drv" = "builtin, native"
"ddraw"        = "builtin, native"
"dinput"       = "builtin, native"
"dsound"       = "builtin, native"
"mmsystem"     = "builtin"
"winmm"        = "builtin"
"msvcrt"       = "native, builtin"
"msvideo"      = "builtin, native"
"msvfw32"      = "builtin, native"
"mcicda.drv"   = "builtin, native"
"mciseq.drv"   = "builtin, native"
"mciwave.drv"  = "builtin, native"
"mciavi.drv"   = "native, builtin"
"mcianim.drv"  = "native, builtin"
"msacm.drv"    = "builtin, native"
"msacm"        = "builtin, native"
"msacm32"      = "builtin, native"
"midimap.drv"  = "builtin, native"
"wnaspi32"     = "builtin"
"icmp"         = "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.

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 "masking" 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-"      <
```

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`

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

Even though it sounds like a lot of sections, these are all closely related. They are all 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"
```

Sounds familiar? 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's 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.

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.

[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.

The [WinMM] Section

[WinMM] is used to define which multimedia drivers have to be loaded. Since those drivers may depend on the multimedia interfaces available on your sustem (OSS, Alsa... to name a few), it's needed to be able to configure which driver has to be loaded.

The content of the section looks like:

```
[WinMM]
"Drivers" = "wineoss.drv"
"WaveMapper" = "msacm.drv"
"MidiMapper" = "midimap.drv"
```

All the keys must be defined:

- The "Drivers" key is a ';' separated list of modules name, each of them containing a low level driver. All those drivers will be loaded when MMSYSTEM/WINMM is started and will provide their inner features.
- The "WaveMapper" represents the name of the module containing the Wave Mapper driver. Only one wave mapper can be defined in the system.
- The "MidiMapper" represents the name of the module containing the Midi Mapper driver. Only one Midi mapper can be defined in the system.

The [AppDefaults] Section

The section is used to overwrite the setting of this file for a special program with different settings. [AppDefaults] is not the real name of the section. The real name consists of the leading word AppDefaults followed by the name of the executable the section is valid for. The end of the section name is the name of the section of the configuration file its values should be overwritten with different settings. The three parts of the section name are separated by two backslashes.

Currently wine supports only overwriting the sections [DllOverrides] and [x11drv].

Here is an example that overwrites the normal settings for a program:

```
;; default settings
[x11drv]
"Managed" = "Y"
"Desktop" = "N"

;; run install in desktop mode
[AppDefaults\\install.exe\\x11drv]
"Managed" = "N"
"Desktop" = "800x600"
```

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. NOTE: this file is currently unused as a new global configuration mechanism is not in place at this time

`$HOME/.wine/config`

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/.wine/config` for wine to recognize it.

What If It Doesn't Work?

There is always a chance that things will go wrong. If the unthinkable happens report the problem to Wine Bugzilla (http://bugs.winehq.com/), try the newsgroup comp.emulators.ms-windows.wine, or the IRCnet channel #WineHQ found on irc.stealth.net:6668, or connected servers. Make sure that you have looked over this document thoroughly, and have also read:

- `README`
- `http://www.la-sorciere.de/wine/index.html` (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.

Win95/98 Look

Written by David A. Cuthbert <`dacut@ece.cmu.edu`>

(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/config` for a `"WineLook" = "Win95"` or `"WineLook" = "Win98"` entry.

A few new sections and a number of entries have been added to the `~/.wine/config` 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 system typeface
"System.Bold" = "[true|false]"      # Whether the system font should be boldfaced
"System.Italic" = "[true|false]"    # Whether the system font should be italicized
"System.Underline" = "[true|false]" # Whether the system font should be underlined
"System.StrikeOut" = "[true|false]" # Whether the system font should be struck out
"OEMFixed.xxx"                  # Same parameters for the OEM fixed typeface
"AnsiFixed.xxx"                 # Same parameters for the Ansi fixed typeface
"AnsiVar.xxx"                   # Same parameters for the Ansi variable typeface
"SystemFixed.xxx"               # Same parameters for the System fixed typeface

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

Configuring the x11drv Driver

Written by Ove Kåven <ovek@winehq.com>

(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/config, but I will only cover the x11drv driver in this article.

x11drv modes of operation

Note: This is now all done in the config file. Needs an update...

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.

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.

TextCP

To be documented...

XVideoPort

To be documented...

Synchronous

To be documented...

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.

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.

Using a Windows registry

If you point Wine at an existing MS Windows installation (by setting the appropriate directories in `~/.wine/config`, 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.

Wine registry data files

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.

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 source 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.

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 in the Wine source, but if you want to install it manually, you can do so by using the **regapi** tool to be found in the `programs/regapi/` directory in Wine source.

The [registry] section

With the above information fresh in mind, let's look at the `wine.conf`/`~/.wine/config` 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 uses 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.

Drive labels and serial numbers with wine

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

Changes by Andreas Mohr <`amohr@codeweavers.com`> 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.

What's Supported?

File System	Types	Comment
FAT systems	hd, floppy	reads labels and serial numbers
ISO9660	cdrom	reads labels and serial numbers (not mixed-mode CDs yet !)

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/config`. 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/config`, 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.

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 really needed here as we have a fixed label
"Device" = "/dev/cdrom"
"Filesystem" = "msdos"
```

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? ....

DLL configuration

DLL Overrides

Written by Ove Kåven <ovek@winehq.com>

(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.

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 have exactly the same API on both Windows and Unix.

The [DllDefaults] section

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.

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 ~/.wine/config, you may safely delete it.

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.

Missing DLLs

Written by Andreas Mohr <`amohr@codeweavers.com`>

In case Wine complains about a missing DLL, you should check whether this file is a publicly available DLL or a custom DLL belonging to your program (by searching for its name on the internet). If you managed to get hold of the DLL, then you should make sure that Wine is able to find and load it. DLLs usually get loaded according to the mechanism of the SearchPath() function. This function searches directories in the following order: a) The directory the program was started from. b) The current directory. c) The Windows system directory. d) The Windows directory. e) The PATH variable directories. In short: either put the required DLL into your application directory (might be ugly), or usually put it into the Windows system directory. Just find out its directory by having a look at the Wine config File variable "System" (which indicates the location of the Windows system directory) and the associated drive entry.

Keyboard

Written by Ove Kåven <`ovek@winehq.com`>

(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 `dlls/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[]={
...
...
 {MAKELANGID(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 dlls/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 dlls/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.

Dealing with Fonts

Fonts

Written by Alex Korobka <`alex@aikea.ams.sunysb.edu`>

(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/development/

How To Convert Windows Fonts

If you have access to a Windows installation you should use the **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**.
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. You may also or instead have to restart the font server (using e.g. **/etc/init.d/xfs restart** under RedHat 7.1)
5. Edit the `~/.wine/config` 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 :-)

How To Add Font Aliases To `~/.wine/config`

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-"
"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] <, optional "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"
-misc-fixed-...	->	"Fixed"

X font	...will show up as...	Extracted name
-...	->	
-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.

How To Manage Cached Font Metrics

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

```
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.[display]` with the new information. This process can take a while.

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.

"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.

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...

Printing in Wine

How to print documents in Wine...

Printing

Written by Huw D M Davies <`h.davies1@physics.ox.ac.uk`>

(Extracted from `wine/documentation/printing`)

Printing in Wine can be done in one of two ways:

1. Use the builtin Wine PostScript driver (+ ghostscript to produce output for non-PostScript printers).
2. Use an external windows 3.1 printer driver (outdated, probably won't get supported any more).

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

Builtin Wine PostScript driver

Enables printing of PostScript files via a driver built into Wine. See below for installation instructions. The code for the PostScript driver is in `dlls/wineps/`.

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.

External printer drivers (non-working as of Jul 8, 01)

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 `~/.wine/config`). 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 `~/.wine/config`. The code for the driver interface is in `graphics/win16drv`.

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.

There are now also virtual spool queues called LPR:printername, which send the data to **lpr -Pprintername**. You do not need to specify those in the config file, they are handled automatically by `dlls/gdi/printdrv.c`.

The Wine PostScript Driver

Written by Huw D M Davies <`h.davies1@physics.ox.ac.uk`>

(Extracted from `wine/documentation/psdriver`)

This allows Wine to generate PostScript files without needing an external printer driver. Wine in this case uses the system provided PostScript printer filters, which almost all use ghostscript if necessary. Those should be configured during the original system installation or by your system administrator.

Installation

Installation of CUPS printers

If you are using CUPS, you do not need to configure .ini or registry entries, everything is autodetected.

Installation of LPR /etc/printcap based printers

If your system is not yet using CUPS, it probably uses LPRng or a LPR based system with configuration based on /etc/printcap.

If it does, your printers in `/etc/printcap` are scanned with a heuristic whether they are PostScript capable printers and also configured mostly automatic.

Since WINE cannot find out what type of printer this is, you need to specify a PPD file in the [ppd] section of `~/.wine/config`. Either use the shortcut name and make the entry look like:

```
[ppd]
"ps1" = "/usr/lib/wine/ps1.ppd"
```

Or you can specify a generic PPD file that is to match for all of the remaining printers. A generic PPD file can be found in `documentation/samples/generic.ppd`.

Installation of other printers

You do not need to do this if the above 2 sections apply, only if you have a special printer.

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

to the [devices] section and

```
Wine PostScript Driver=WINEPS,LPT1:,15,45
```

to the [PrinterPorts] 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`.

You also need to add certain entries to the registry. The easiest way to do this is to customise the contents of `documentation/psdrv.reg` (see below) and use the Winelib program **programs/regapi/regapi**. For example, if you have installed the Wine source tree in `/usr/src/wine`, you could use the following series of commands:

- **`cp /usr/src/wine/documentation/psdrv.reg ~`**
- **`vi ~/psdrv.reg`**
- Edit the copy of `psdrv.reg` to suit your requirements. At a minimum, you must specify a PPD file for each printer.
- **`regapi setValue < ~/psdrv.reg`**

Required configuration for all printer types

You won't need Adobe Font Metric (AFM) files for the (type 1 PostScript) fonts that you wish to use any more. Wine now has this information builtin.

You'll need 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 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/). See above for information on configuring the driver to use this file.

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 the builtin PostScript driver.

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.

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 D M Davies <h.davies1@physics.ox.ac.uk>

Chapter 5. Running Wine

Written by John R. Sheets <jsheets@codeweavers.com>

How to run Wine

Wine is a very complicated piece of software with many ways to adjust how it runs. With very few exceptions, you can activate the same set of features through the configuration file as you can with command-line parameters. In this chapter, we'll briefly discuss these parameters, and match them up with their corresponding configuration variables.

You can invoke the **wine --help** command to get a listing of all Wine's command-line parameters:

```
Usage: ./wine [options] program_name [arguments]

Options:
   --debugmsg name  Turn debugging-messages on or off
   --dll name       Enable or disable built-in DLLs
   --dosver x.xx    DOS version to imitate (e.g. 6.22)
                    Only valid with --winver win31
   --help,-h        Show this help message
   --managed        Allow the window manager to manage created windows
   --version,-v     Display the Wine version
   --winver         Version to imitate
                    (win95,nt40,win31,nt2k,win98,nt351,win30,win20)
```

You can specify as many options as you want, if any. Typically, you will want to have your configuration file set up with a sensible set of defaults; in this case, you can run **wine** without explicitly listing any options. In rare cases, you might want to override certain parameters on the command line.

After the options, you should put the name of the file you want **wine** to execute. If the executable is in the *Path* parameter in the configuration file, you can simply give the executable file name. However, if the executable is not in *Path*, you must give the full path to the executable (in Windows format, not UNIX format!). For example, given a *Path* of the following:

```
[wine]
"Path"="c:\windows;c:\windows\system;e:\;e:\test;f:\"
```

You could run the file `c:\windows\system\foo.exe` with:

```
$ wine foo.exe
```

However, you would have to run the file `c:\myapps\foo.exe` with this command:

```
$ wine c:\myapps\foo.exe
```

Finally, if you want to pass any parameters to your windows application, you can list them at the end, just after the executable name. Thus, to run the imaginary **foo.exe** Windows application with its */advanced* mode parameter, while invoking Wine in *--managed* mode, you would do something like this:

```
$ wine --managed foo.exe /advanced
```

In other words, options that affect Wine should come *before* the Windows program name, while options that affect the Windows program should come *after* it.

If you want to run a console program (aka a CUI executable), use **wineconsole** instead of **wine** to start it. It will display the program in a separate Window (this requires X11 to be run). If you don't, you'll still be able to run able your program, in the Unix console were you're started your program, but with very limited capacities (so, your program might work, but your mileage may vary). This shall be improved in the future.

Command-Line Options

--debugmsg [channels]

Wine isn't perfect, and many Windows applications still don't run without bugs under Wine (but then, many of them don't run without bugs under native Windows either!). To make it easier for people to track down the causes behind each bug, Wine provides a number of *debug channels* that you can tap into.

Each debug channel, when activated, will trigger logging messages to be displayed to the console where you invoked **wine**. From there you can redirect the messages to a file and examine it at your leisure. But be forewarned! Some debug channels can generate incredible volumes of log messages. Among the most prolific offenders are `relay` which spits out a log message every time a win32 function is called, `win` which tracks windows message passing, and of course `all` which is an alias for every single debug channel that exists. For a complex application, your debug logs can easily top 1 MB and higher. A `relay` trace can often generate more than 10 MB of log messages, depending on how long you run the application. Logging does slow down Wine quite a bit, so don't use `--debugmsg` unless you really do want log files.

Within each debug channel, you can further specify a *message class*, to filter out the different severities of errors. The four message classes are: `trace`, `fixme`, `warn`, `err`.

To turn on a debug channel, use the form `class+channel`. To turn it off, use `class-channel`. To list more than one channel in the same `--debugmsg` option, separate them with commas. For example, to request `warn` class messages in the `heap` debug channel, you could invoke **wine** like this:

```
$ wine --debugmsg warn+heap program_name
```

If you leave off the message class, **wine** will display messages from all four classes for that channel:

```
$ wine --debugmsg +heap program_name
```

If you wanted to see log messages for everything except the relay channel, you might do something like this:

```
$ wine --debugmsg +all,-relay program_name
```

Here is a master list of all the debug channels and classes in Wine. More channels might be added to (or subtracted from) later versions.

Table 5-1. Debug Channels

all	accel	advapi	animate	aspi
atom	avifile	bitblt	bitmap	caret
cdrom	class	clipboard	clipping	combo
comboex	comm	commctrl	commdlg	console
crtdll	cursor	datetime	dc	ddeml
ddraw	debug	debugstr	delayhlp	dialog
dinput	dll	dosfs	dosmem	dplay
driver	dsound	edit	elfdll	enhmetafile
event	exec	file	fixup	font

gdi	global	graphics	header	heap
hook	hotkey	icmp	icon	imagehlp
imagelist	imm	int	int10	int16
int17	int19	int21	int31	io
ipaddress	joystick	key	keyboard	loaddll
ldt	listbox	listview	local	mci
mcianim	mciavi	mcicda	mcimidi	mciwave
mdi	menu	message	metafile	midi
mmaux	mmio	mmsys	mmtime	module
monthcal	mpr	msacm	msg	msvideo
nativefont	nonclient	ntdll	odbc	ole
opengl	pager	palette	pidl	print
process	profile	progress	prop	propsheet
psapi	psdrv	ras	rebar	reg
region	relay	resource	richedit	scroll
segment	seh	selector	sendmsg	server
setupapi	setupx	shell	snoop	sound
static	statusbar	storage	stress	string
syscolor	system	tab	tape	tapi
task	text	thread	thunk	timer
toolbar	toolhelp	tooltips	trackbar	treeview
ttydrv	tweak	typelib	updown	ver
virtual	vxd	wave	win	win16drv
win32	winedbg	wing	wininet	winsock
winspool	wnet	x11		

For more details about debug channels, check out the The Wine Developer's Guide (http://wine.codeweavers.com/docs/wine-devel/).

--dll

--dosver

--help

--managed

--version

--winver

Chapter 6. Finding and Reporting Bugs

How To Report A Bug

Written by (???)

(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.

With using either approach report the found issues with relevant Wine Bugzilla (http://bugs.winehq.com/).

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 the bug to Wine Bugzilla (http://bugs.winehq.com/). Please, search Bugzilla database to check whether your problem is already found before posting a bug report. Include your own detailed description of the problem with relevant information. Attach the "Nice Formatted Report" to the submitted bug. Do not cut and paste the report in the bug description - it is pretty big. Keep the full debug output in case it will be needed by Wine developers.

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 used with Wine. Mention if you don't use Windows
   - 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! You can also turn on other parameters, depending on the nature of the problem you are researching. See wine man page for full list of the parameters.

To get the trace output, use the following commands:

all shells:

```
    $ echo quit | wine -debugmsg +relay [other_options] program_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] program_name |& tee filename.out;
$ tail -100 filename.out > report_file
```

bash and other sh-like shells:

```
    $ wine -debugmsg +relay [other_options] program_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 the bug to Wine Bugzilla (http://bugs.winehq.com/). Please, search Bugzilla database to check whether your problem is

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

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 Wine Bugzilla (http://bugs.winehq.com/) to let us know how this document can be improved.

Wine Developer’s Guide

Wine Developer’s Guide

Table of Contents

I. Developing Wine

Chapter 1. Compiling Wine

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

Compiling Wine

Tools required

- gcc -- 2.7.x required (Wine uses attribute stdcall). Versions earlier than 2.7.2.3 barf on shellord.c -- compile without optimizing for that file. In addition EGCS 1.1.x and GCC 2.95.x are reported to work fine.
- flex >= 2.5.1 (required for the debugger and wrc, and lex won't do)
- bison (also required for debugger. Don't know whether BSD yacc would work.)
- X11 libs and include files
- texinfo >= 3.11 (optional, to compile the documentation.)
- autoconf (if you want to remake configure, which is not normally required)
- XF86DGA extension (optional, detected by configure, needed for DirectX support)
- Open Sound System (optional, detected by configure, for sound support)

The Red Hat RPMs are gcc-XXX, flex-XXX, and XFree86-devel-XXX, where XXX is the version number.

Space required

You also need about 230 MB of available disk space for compilation. The compiled libwine.so binary takes around 5 MB of disk space, which can be reduced to about 1 MB by stripping ('strip wine'). Stripping is not recommended, however, as you can't submit proper crash reports with a stripped binary any more.

Common problems

If you get a repeatable sig11 compiling shellord.c, thunk.c or other files, try compiling just that file without optimization. Then you should be able to finish the build.

OS specific issues

- FreeBSD -- In order to run Wine, the FreeBSD kernel needs to be compiled with

options	USER_LDT
options	SYSVSHM
options	SYSVSEM
options	SYSVMSG

  If you need help, read the chapter "Building and Installing a Custom Kernel (http://www.freebsd.org/handbook/kernelconfig-building.html)" in the "FreeBSD handbook (http://www.freebsd.org/handbook/). You'll need to be running FreeBSD 3.x or later.

- SCO Unixware, Openserver -- UW port is supported by SCO.

- OS/2 -- not a complete port. See Odin (http://odin.netlabs.org/ProjectAbout.phtml) for a project which uses some Wine code.
- Solaris x86 2.x -- Needs GNU toolchain (gcc, gas, flex as above, yacc may work) to compile, seems functional (980215).
- DGUX, HP, Irix, or other Unixes; non-intel Linux. No ports have been seriously attempted. For non-intel Unixes, only a winelib port is relevant. Alignment may be a problem.
- Macintosh/Rhapsody/BeOS -- no ports have been attempted.

Chapter 2. Debugging Wine

Introduction

Written by Eric Pouech <Eric.Pouech@wanadoo.fr> (Last updated: 6/14/2000)

(Extracted from wine/documentation/winedbg)

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`).

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 of 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's modes of invocation

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"
```

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

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`.

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**.

Using the Wine Debugger

Written by Marcus Meissner <Marcus.Meissner@caldera.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.

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)
|1 0x01b7:0x1e5b (PXSRV_FONPUTCATFONT+0x2cd)
|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.

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.

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".

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.

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
|CallTo16(func=01d7:001a,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,0x0000,0x0000,0x0000,0x0000,0x0800,0x0000,0x0000,0x0dcf)
|...                                                   <----- 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: GETTEMPFILENAME(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 returns '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: OPENFILE(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 verlassen und SHARE.EXE laden bevor
```

And MessageBox'ed.

```
|Stopped on breakpoint 2 at 0x40189100 (MessageBoxA [msgbox.c:190])
|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.

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 error 219 at 0004:1056.",0x00000000,0x1010)
  ```

  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: WINHELP(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 error 219 at 0004:1056.",0x00000000,0x1010)
```

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.

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.

Useful memory addresses

Written by Andreas Mohr <amohr@codeweavers.com>

Wine uses several different kinds of memory addresses.

Win32/"normal" Wine addresses/Linux: linear addresses.

Linear addresses can be everything from 0x0 up to 0xffffffff. In Wine on Linux they are often around e.g. 0x08000000, 0x00400000 (std. Win32 program load address), 0x40000000. Every Win32 process has its own private 4GB address space (that is, from 0x0 up to 0xffffffff).

Win16 "enhanced mode": segmented addresses.

These are the "normal" Win16 addresses, called SEGPTR. They have a segment:offset notation, e.g. 0x01d7:0x0012. The segment part usually is a "selector", which *always* has the lowest 3 bits set. Some sample selectors are 0x1f7, 0x16f, 0x8f. If these bits are set except for the lowest bit, as e.g. with 0x1f6,xi then it might be a handle to global memory. Just set the lowest bit to get the selector in these cases. A selector kind of "points" to a certain linear (see above) base address. It has more or less three important attributes: segment base address, segment limit, segment access rights.

Example:

Selector 0x1f7 (0x40320000, 0x0000ffff, r-x) So 0x1f7 has a base address of 0x40320000, the segment's last address is 0x4032ffff (limit 0xffff), and it's readable and executable. So an address of 0x1f7:0x2300 would be the linear address of 0x40322300.

DOS/Win16 "standard mode"

They, too, have a segment:offset notation. But they are completely different from "normal" Win16 addresses, as they just represent at most 1MB of memory: The segment part can be anything from 0 to 0xffff, and it's the same with the offset part.

Now the strange thing is the calculation that's behind these addresses: Just calculate segment*16 + offset in order to get a "linear DOS" address. So e.g. 0x0f04:0x3628 results in 0xf040 + 0x3628 = 0x12668. And the highest address you can get is 0xfffff (1MB), of course. In Wine, this "linear DOS" address of 0x12668 has to be added to the linear base address of the corresponding DOS memory allocated for dosmod in order to get the true linear address of a DOS seg:offs address. And make sure that you're doing this in the correct process with the correct linear address space, of course ;-)

Configuration

Registry configuration

The Windows' debugging API uses a registry entry to know which debugger to invoke when an unhandled exception occurs (see *On exception* 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). The path to the debugger you chose to use must be reachable via a DOS drive in the Wine config file !

You can also set a shell script to launch the debugger. In this case, you need to be sure that the invocation in this shell script is of the form:

```
WINEPRELOAD=<path_to_winedbg.so> exec wine $*
```

(Shell script must use exec, and the debugger .so file must be preloaded to override the shell script information).

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 (available in Wine source) 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 configuration

WineDbg can be configured through 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 these options, 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:

Controlling 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 through 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).

`BreakOnDllLoad`

Set to `TRUE` if the debugger stops when a DLL is loaded into memory; when the debugger is invoked after a crash, the DLLs already mapped in memory will not trigger this break. `FALSE` by default.

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.

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.

WineDbg Command Reference

Misc

```
abort aborts the debugger
quit exits the debugger

attach N attach to a W-process (N is its ID). IDs can be
obtained using the walk process command
detach          detach from a W-process. WineDbg will exit (this may
                be changed later on)


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
```

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.

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)
```

When setting a breakpoint on an <id>, if several symbols with this <id> exist, the debugger will prompt for the symbol you want to use. Pick up the one you want from its number.

Alternatively you can specify a DLL in the <id> (for example MYDLL.DLL.myFunc for function myFunc of `G:\AnyPath\MyDll.dll`).

You can use the symbol *EntryPoint* to stand for the entry point of the Dll.

Stack manipulation

```
bt print calling stack of current thread
bt N print calling stack of thread of ID N (note: this
                doesn't change the position of the current frame as
manipulated by the up & dn commands)
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 for current thread's stack
info local prints information on local variables for current
function
```

Directory & source file manipulation

```
show dir
dir <pathname>
dir
symbolfile <module> <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
```

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)
```

Disassembly

```
disas disassemble from current position
disas <expr> disassemble from address <expr>
disas <expr>,<expr>disassembles code between addresses specified by
the two <expr>
```

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)
```

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
```

Expressions

Expressions in Wine Debugger are mostly written in a C form. However, there are a few discrepancies:

- Identifiers can take a '.' in their names. This allow mainly to access symbols from different DLLs like USER32.DLL.CreateWindowA
- The debugger will try to distinguish this writing with structure operations. Therefore, you can only use the previous writing in operations manipulating symbols ({break|watch}points, type information command...).

Other debuggers

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.

Following text written by Andreas Mohr <`amohr@codeweavers.com`>

Here's how to get info about the current execution status of a certain Wine process:

Change into your Wine source dir and enter:

```
$ gdb wine
```

Switch to another console and enter **ps ax | grep wine** to find all wine processes. Inside **gdb**, repeat for all Wine processes:

```
(gdb) attach PID
```

with **`PID`** being the process ID of one of the Wine processes. Use

```
(gdb) bt
```

to get the backtrace of the current Wine process, i.e. the function call history. That way you can find out what the current process is doing right now. And then you can use several times:

```
(gdb) n
```

or maybe even

```
(gdb) b SomeFunction
```

and

```
(gdb) c
```

to set a breakpoint at a certain function and continue up to that function. Finally you can enter

```
(gdb) detach
```

to detach from the Wine process.

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.

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)

Limitations

16 bit processes are not supported (but calls to 16 bit code in 32 bit applications are).

Chapter 3. Documenting Wine

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

Writing Wine API Documentation

Written by Douglas Ridgway <ridgway@winehq.com>

(Extracted from wine/documentation/README.documentation)

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

```
/***********************************************************
 *         CopyMetaFileA   (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.
 */
HMETAFILE WINAPI CopyMetaFileA(
   HMETAFILE 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   (GDI32.23)

SYNOPSIS
       HMETAFILE CopyMetaFileA
       (
            HMETAFILE hSrcMetaFile,
            LPCSTR lpFilename
       );

PARAMETERS
       HMETAFILE 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)
```

The Wine DocBook System

Written by John R. Sheets <jsheets@codeweavers.com>

Writing Documentation with DocBook

DocBook is a flavor of SGML (*Standard Generalized Markup Language*), a syntax for marking up the contents of documents. HTML is another very common flavor of SGML; DocBook markup looks very similar to HTML markup, although the names of the markup tags differ.

Terminology

SGML markup contains a number of syntactical elements that serve different purposes in the markup. We'll run through the basics here to make sure we're on the same page when we refer to SGML semantics.

The basic currency of SGML is the *tag*. A simple tag consists of a pair of angle brackets and the name of the tag. For example, the `para` tag would appear in an SGML document as `<para>`. This start tag indicates that the immediately following text should be classified according to the tag. In regular SGML, each opening tag must have a matching end tag to show where the start tag's contents end. End tags begin with "`</`" markup, e.g., `</para>`.

The combination of a start tag, contents, and an end tag is called an *element*. SGML elements can be nested inside of each other, or contain only text, or may be a combination of both text and other elements, although in most cases it is better to limit your elements to one or the other.

The XML (*eXtensible Markup Language*) specification, a modern subset of the SGML specification, adds a so-called *empty tag*, for elements that contain no text content. The entire element is a single tag, ending with "`/>`", e.g., `<xref/>`. However, use of this tag style restricts you to XML DocBook processing, and your document may no longer compile with SGML-only processing systems.

Often a processing system will need more information about an element than you can provide with just tags. SGML allows you to add extra "hints" in the form of SGML *attributes* to pass along this information. The most common use of attributes in DocBook is giving specific elements a name, or an ID, so you can refer to it from elsewhere. This ID can be used for many things, including file-naming for HTML output, hyper-linking to specific parts of the document, and even pulling text from that element (see the `<xref>` tag).

An SGML attribute appears inside the start tag, between the < and > brackets. For example, if you wanted to set the `id` attribute of the `<book>` element to "mybook", you would create a start tag like this:

```
<book id="mybook">
```

Notice that the contents of the attribute are enclosed in quote marks. These quotes are optional in SGML, but mandatory in XML. It's a good habit to use quotes, as it will make it much easier to migrate your documents to an XML processing system later on.

You can also specify more than one attribute in a single tag:

```
<book id="mybook" status="draft">
```

Another commonly used type of SGML markup is the *entity*. An entity lets you associate a block of text with a name. You declare the entity once, at the beginning of your document, and can invoke it as many times as you like throughout the document. You can use entities as shorthand, or to make it easier to maintain certain phrases in a central location, or even to insert the contents of an entire file into your document.

An entity in your document is always surrounded by the "&" and ";" characters. One entity you'll need sooner or later is the one for the "<" character. Since SGML expects all tags to begin with a "<", the "<" is a reserved character. To use it in your document (as I am doing here), you must insert it with the `<` entity. Each time the SGML processor encounters `<`, it will place a literal "<" in the output document. Similarly you must use the `>` and `&` entities for the ">" and "&" characters.

The final term you'll need to know when writing simple DocBook documents is the DTD (*Document Type Declaration*). The DTD defines the flavor of SGML a given document is written in. It lists all the legal tag names, like `<book>`, `<para>`, and so on, and declares how those tags are allowed to be used together. For example, it doesn't make sense to put a `<book>` element inside a `<para>` paragraph element -- only the reverse.

The DTD thus defines the legal structure of the document. It also declares which attributes can be used with which tags. The SGML processing system can use the DTD to make sure the document is laid out properly before attempting to process it. SGML-aware text editors like Emacs can also use the DTD to guide you while you write, offering you choices about which tags you can add in different places in the document, and beeping at you when you try to add a tag where it doesn't belong.

Generally, you will declare which DTD you want to use as the first line of your SGML document. In the case of DocBook, you will use something like this:

```
<!doctype book PUBLIC "-//OASIS//DTD
           DocBook V3.1//EN" []> <book> ...
           </book>
```

Note that you must specify your toplevel element inside the doctype declaration. If you were writing an article rather than a book, you might use this declaration instead:

```
<!doctype article PUBLIC "-//OASIS//DTD DocBook V3.1//EN" []>
<article>
...
</article>
```

The Document

Once you're comfortable with SGML, creating a DocBook document is quite simple and straightforward. Even though DocBook contains over 300 different tags, you can usually get by with only a small subset of those tags. Most of them are for inline formatting, rather than for document structuring. Furthermore, the common tags have short, intuitive names.

Below is a (completely nonsensical) example to illustrate how a simple document might be laid out. Notice that all `<chapter>` and `<sect1>` elements have `id` attributes. This is not mandatory, but is a good habit to get into, as DocBook is commonly converted into HTML, with a separate generated file for each `<book>`, `<chapter>`, and/or `<sect1>` element. If the given element has an `id` attribute, the processor will typically name the file accordingly. Thus, the below document might result in `index.html`, `chapter-one.html`, `blobs.html`, and so on.

Also notice the text marked off with “<!-- ” and “ -->” characters. These denote SGML comments. SGML processors will completely ignore anything between these markers, similar to “/*” and “*/” comments in C source code.

```
<!doctype book PUBLIC "-//OASIS//DTD DocBook V3.1//EN" []>
<book id="index">
  <bookinfo>
    <title>A Poet’s Guide to Nonsense</title>
  </bookinfo>

  <chapter id="chapter-one">
    <title>Blobs and Gribbles</title>

    <!-- This section contains only one major topic -->
    <sect1 id="blobs">
      <title>The Story Behind Blobs</title>
      <para>
        Blobs are often mistaken for ice cubes and rain
        puddles...
      </para>
    </sect1>

    <!-- This section contains embedded sub-sections -->
    <sect1 id="gribbles">
      <title>Your Friend the Gribble</title>
      <para>
        A Gribble is a cute, unassuming little fellow...
      </para>

      <sect2 id="gribble-temperament">
        <title>Gribble Temperament</title>
        <para>
          When left without food for several days...
        </para>
      </sect2>

      <sect2 id="gribble-appearance">
        <title>Gribble Appearance</title>
        <para>
          Most Gribbles have a shock of white fur running from...
        </para>
      </sect2>
    </sect1>
  </chapter>

  <chapter id="chapter-two">
    <title>Phantasmagoria</title>

    <sect1 id="dretch-pools">
      <title>Dretch Pools</title>

      <para>
        When most poets think of Dretch Pools, they tend to...
      </para>
    </sect>
  </chapter>
</book>
```

Common Elements

Once you get used to the syntax of SGML, the next hurdle in writing DocBook documentation is to learn the many DocBook-specific tag names, and when to use them. DocBook was created for technical documentation, and as such, the tag names and document structure are slanted towards the needs of such documentation.

To cover its target audience, DocBook declares a wide variety of specialized tags, including tags for formatting source code (with somewhat of a C/C++ bias), computer prompts, GUI application features, keystrokes, and so on. DocBook also includes tags for universal formatting needs, like headers, footnotes, tables, and graphics.

We won't cover all of these elements here (over 300 DocBook tags exist!), but we will cover the basics. To learn more about the other tags, check out the official DocBook guide, at http://docbook.org. To see how they are used in practice, download the SGML source for this manual (the Wine Developer Guide) and browse through it, comparing it to the generated HTML (or PostScript or PDF).

There are often many correct ways to mark up a given piece of text, and you may have to make guesses about which tag to use. Sometimes you'll have to make compromises. However, remember that it is possible to further customize the output of the SGML processors. If you don't like the way a certain tag looks in HTML, that doesn't mean you should choose a different tag based on its output formatting. The processing stylesheets can be altered to fix the formatting of that same tag everywhere in the document (not just in the place you're working on). For example, if you're frustrated that the `<systemitem>` tag doesn't produce any formatting by default, you should fix the stylesheets, not change the valid `<systemitem>` tag to, for example, an `<emphasis>` tag.

Here are the common SGML elements:

Structural Elements

`<book>`

The book is the most common toplevel element, and is probably the one you should use for your document.

`<set>`

If you want to group more than one book into a single unit, you can place them all inside a set. This is useful when you want to bundle up documentation in alternate ways. We do this with the Wine documentation, using a `<set>` to put everything into a single directory (see `documentation/wine-doc.sgml`), and a `<book>` to put each Wine guide into a separate directory (see `documentation/wine-devel.sgml`, etc.).

`<chapter>`

A `<chapter>` element includes a single entire chapter of the book.

`<part>`

If the chapters in your book fall into major categories or groupings (as in the Wine Developer Guide), you can place each collection of chapters into a `<part>` element.

`<sect?>`

DocBook has many section elements to divide the contents of a chapter into smaller chunks. The encouraged approach is to use the numbered section tags, `<sect1>`, `<sect2>`, `<sect3>`, `<sect4>`, and `<sect5>` (if necessary). These tags must be nested in order: you can't place a `<sect3>` directly inside a `<sect1>`. You have to nest the `<sect3>` inside a `<sect2>`, and so forth. Documents with these explicit section groupings are easier for SGML processors to deal with, and lead to better organized documents. DocBook also supplies a `<section>` element which you can nest inside itself, but its use is discouraged in favor of the numbered section tags.

`<title>`

The title of a book, chapter, part, section, etc. In most of the major structural elements, like `<chapter>`, `<part>`, and the various section tags, `<title>` is mandatory. In other elements like `<book>` and `<note>`, it's optional.

`<para>`

The basic unit of text is the paragraph, represented by the `<para>` tag. This is probably the tag you'll use most often. In fact, in a simple document, you can probably get away with using only `<book>`, `<chapter>`, `<title>`, and `<para>`.

`<article>`

For shorter, more targeted documents, like topic pieces and whitepapers, you can use `<article>` as your toplevel element.

Inline Formatting Elements

`<filename>`

The name of a file. You can optionally set the `class` attribute to `Directory`, `HeaderFile`, and `SymLink` to further classify the filename.

`<userinput>`

Literal text entered by the user.

`<computeroutput>`

Literal text output by the computer.

`<literal>`

A catch-all element for literal computer data. Its use is somewhat vague; try to use a more specific tag if possible, like `<userinput>` or `<computeroutput>`.

`<quote>`

An inline quotation. This tag typically inserts quotation marks for you, so you would write `<quote>`This is a quote`</quote>` rather than "This is a quote". This usage may be a little bulkier, but it does allow for automated formatting of all quoted material in the document. Thus, if you wanted all quotations to appear in italic, you could make the change once in your stylesheet, rather than doing a search and replace throughout the document. For larger chunks of quoted text, you can use `<blockquote>`.

`<note>`

Insert a side note for the reader. By default, the SGML processor usually prefixes the content with "Note:". You can change this text by adding a `<title>` element. Thus, to add a visible FIXME comment to the documentation, you might write:

```
<note>
  <title>FIXME</title>
  <para>This section needs more info about...</para>
</note>
```

The results will look something like this:

FIXME: This section needs more info about...

`<sgmltag>`

Used for inserting SGML tags, etc., into a SGML document without resorting to a lot of entity quoting, e.g., <. You can change the appearance of the text with the `class` attribute. Some common values of this are `starttag`, `endtag`, `attribute`, `attvalue`, and even `sgmlcomment`. See this SGML file, `documentation/documentation.sgml`, for examples.

`<prompt>`

The text used for a computer prompt, for example a shell prompt, or command-line application prompt.

`<replaceable>`

Meta-text that should be replaced by the user, not typed in literally, e.g., in command descriptions and *`--help`* outputs.

`<constant>`

A programming constant, e.g., `MAX_PATH`.

`<symbol>`

A symbolic value replaced, for example, by a pre-processor. This applies primarily to C macros, but may have other uses. Use the `<constant>` tag instead of `<symbol>` where appropriate.

`<function>`

A programming function name.

`<parameter>`

Programming language parameters you pass with a function.

`<option>`

Parameters you pass to a command-line executable.

`<varname>`

Variable name, typically in a programming language.

`<type>`

Programming language types, e.g., from a typedef definition. May have other uses, too.

`<structname>`

The name of a C-language struct declaration, e.g., sockaddr.

`<structfield>`

A field inside a C struct.

`<command>`

An executable binary, e.g., **wine** or **ls**.

`<envar>`

An environment variable, e.g, $PATH.

`<systemitem>`

A generic catch-all for system-related things, like OS names, computer names, system resources, etc.

`<email>`

An email address. The SGML processor will typically add extra formatting characters, and even a `mailto:` link for HTML pages. Usage: `<email>`user@host.com`</email>`

`<firstterm>`

Special emphasis for introducing a new term. Can also be linked to a `<glossary>` entry, if desired.

Item Listing Elements

`<itemizedlist>`

For bulleted lists, no numbering. You can tweak the layout with SGML attributes.

`<orderedlist>`

A numbered list; the SGML processor will insert the numbers for you. You can suggest numbering styles with the `numeration` attribute.

`<simplelist>`

A very simple list of items, often inlined. Control the layout with the `type` attribute.

`<variablelist>`

A list of terms with definitions or descriptions, like this very list!

Block Text Quoting Elements

`<programlisting>`

Quote a block of source code. Typically highlighted in the output and set off from normal text.

`<screen>`

Quote a block of visible computer output, like the output of a command or chunks of debug logs.

Hyperlink Elements

`<link>`

Generic hypertext link, used for pointing to other sections within the current document. You supply the visible text for the link, plus the name of the `id` attribute of the element that you want to link to. For example:

```
<link linkend="configuring-wine">the section on configuring wine</link>
...
<sect2 id="configuring-wine">
...
```

`<xref>`

In-document hyperlink that can generate its own text. Similar to the `<link>` tag, you use the `linkend` attribute to specify which target element you want to jump to:

```
<xref linkend="configuring-wine">
...
<sect2 id="configuring-wine">
...
```

By default, most SGML processors will autogenerate some generic text for the `<xref>` link, like “Section 2.3.1”. You can use the `endterm` attribute to grab the visible text content of the hyperlink from another element:

```
<xref linkend="configuring-wine" endterm="config-title">
...
<sect2 id="configuring-wine">
  <title id="config-title">Configuring Wine</title>
...
```

This would create a link to the configuring-wine element, displaying the text of the config-title element for the hyperlink. Most often, you’ll add an `id` attribute to the `<title>` of the section you’re linking to, as above, in which case the SGML processor will use the target’s title text for the link text.

Alternatively, you can use an `xreflabel` attribute in the target element tag to specify the link text:

```
<sect1 id="configuring-wine" xreflabel="Configuring Wine">
```

> **Note:** <xref> is an empty element. You don't need a closing tag for it (this is defined in the DTD). In SGML documents, you should use the form <xref>, while in XML documents you should use <xref/>.

`<anchor>`

An invisible tag, used for inserting `id` attributes into a document to link to arbitrary places (i.e., when it's not close enough to link to the top of an element).

`<ulink>`

Hyperlink in URL form, e.g., http://www.winehq.com.

`<olink>`

Indirect hyperlink; can be used for linking to external documents. Not often used in practice.

Multiple SGML files

How to split an SGML document into multiple files...

The SGML Environment

You can write SGML/DocBook documents in any text editor you might find (although as we'll find in the Section called *PSGML Mode in Emacs*, some editors are more friendly for this task than others). However, if you want to convert those documents into a more friendly form for reading, such as HTML, PostScript, or PDF, you will need a working SGML environment. This section attempts to lay out the various SGML rendering systems, and how they are set up on the popular Linux distributions.

DSSSL Environment

Explain tools and methodologies..

XSLT Environment

Explain tools and methodologies...

SGML on Redhat

Most Linux distributions have everything you need already bundled up in package form. Unfortunately, each distribution seems to handle its SGML environment differently, installing it into different paths, and naming its packages according to its own whims.

The following packages seems to be sufficient for RedHat 7.1. You will want to be careful about the order in which you install the rpms.

- sgml-common-*.rpm
- openjade-*.rpm
- perl-SGMLSpm-*.rpm
- docbook-dtd*.rpm
- docbook-style-dsssl-*.rpm
- tetex-*.rpm
- jadetex-*.rpm

- docbook-utils-*.rpm

You can also use ghostscript to view the ps format output and Adobe Acrobat 4 to view the pdf file.

SGML on Debian

List package names and install locations...

SGML on Other Distributions

List package names and install locations...

PSGML Mode in Emacs

Although you can write SGML documentation in any simple text editor, some editors provide extra support for entering SGML tags, and for verifying that the SGML you create is valid. SGML has been around for a long time, and many commercial editors exist for it; however, until recently open source SGML editors have been scarce.

> **FIXME:** List the available commercial and open source SGML editors.

The most commonly used open source SGML editor is Emacs, with the PSGML *mode*, or extension. Emacs does not supply a GUI or WYSIWYG (What You See Is What You Get) interface, but it does provide many helpful shortcuts for creating SGML, as well as automatic formatting, validity checking, and the ability to create your own macros to simplify complex, repetitive actions. We'll touch briefly on each of these points.

The first thing you need is a working installation of Emacs (or XEmacs), with the PSGML package. Most Linux distributions provide both as easy-to-install packages.

Next, you'll need a working SGML environment. See the Section called *The SGML Environment* for more info on setting that up.

The DocBook Build System

Basic Infrastructure

How the build/make system works (makefiles, db2html, db2html-winehq, jade, stylesheets).

Tweaking the DSSSL stylesheets

Things you can tweak, and how to do it (examples from default.dsl and winehq.dsl).

Generating docs for Wine web sites

Explain make_winehq, rsync, etc.

Chapter 4. Submitting Patches

Written by Albert den Haan <>

Patch Format

Your patch should include:

- a description of what was wrong and what is now better (and now broken :).
- your contact information (Name/Handle and e-mail)
- the patch in **diff -u** format (it happens...)

cvs diff -u works great for the common case where a file is edited. However, if you add or remove a file **cvs diff** will not report that correctly so make sure you explicitly take care of this rare case.

For additions: mention that you have some new files and include them as either separate attachments or by appending the **diff -u /dev/null /my/new/file** output of them to any **cvs diff -u** output you may have. Alernatively, use **diff -Nu olddir/ newdir/** in case of multiple new files to add.

For removals, list the files.

Quality Assurance

(Or, "How do I get Alexandre to apply my patch quickly so I can build on it and it will not go stale?")

Make sure your patch applies to the current CVS head revisions. If a bunch of patches are commited to CVS that may affect whether your patch will apply cleanly then verify that your patch does apply! **cvs update** is your friend!

Save yourself some embarasment and run your patched code against more than just your current test example. Experience will tell you how much effort to apply here.

Chapter 5. Internationalization

Adding New Languages

Written by Morten Welinder <>, January 1996.

- Thereafter revised February 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	Slovak	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 6. Tools

bin2res

Written by Juergen Schmied <juergen.schmied@metronet.de> (11/99)

(Extracted from wine/documentation/resources)

This document desribes tools for handling resources within wine

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.

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.

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.

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.

II. Wine Architecture

Chapter 7. Overview

Brief overview of Wine's architecture...

Basic Overview

Written by Ove Kåven <ovek@winehq.com>

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.

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.

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 expect 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.

The Wine server

The Wine server 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 as a client having sent a command, or a wait condition having 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 clients' 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 it to the client to close the duplicate when the client has finished with it).

The Service Thread

The Wine server 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`.

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.

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.

Module Overview

written by (???)

(Extracted from `wine/documentation/internals`)

KERNEL Module

Needs some content...

GDI Module

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.

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.

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.

```
Desktop window - root window
 |     \      '-.
 |      \        '-.
popup -> wnd1  ->  wnd2 - top level windows
 |       \   '-.      '-.
 |        \     '-.      '-.
child1  child2 -> child3  child4     - child windows
```

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 buttons).

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.

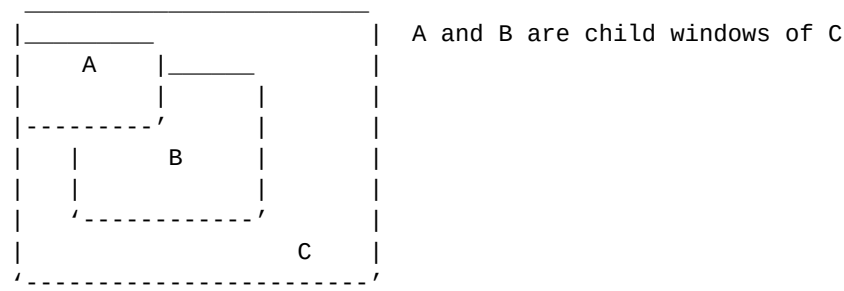
Wine specifics: in default and managed mode each top-level window gets its own X counterpart with desktop window being basically a fake stub. In desktop mode, however, only desktop window has an X window associated with it. Also, `SetWindowPos()` should eventually be implemented via `Begin/End/DeferWindowPos()` calls and not the other way around.

Visible region, clipping region and update region

`windows/dce.c`

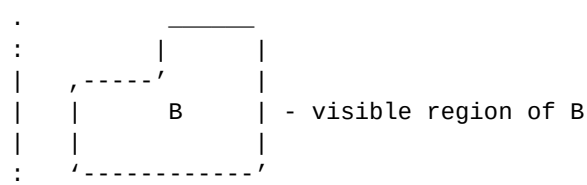
`windows/winpos.c`

`windows/painting.c`



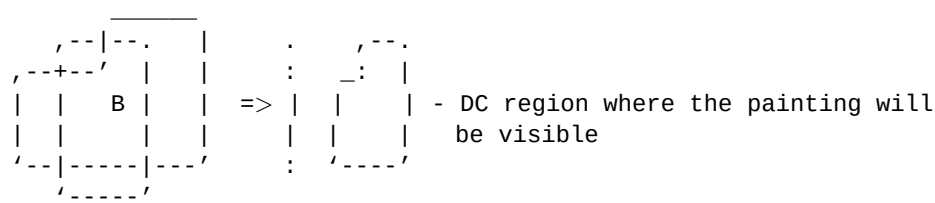
Visible region determines which part of the window is not obscured by other windows. If a window has the `WS_CLIPCHILDREN` style then all areas below its children are considered invisible. Similarily, if the `WS_CLIPSIBLINGS` bit is in effect then all areas obscured by its siblings are invisible. Child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

B has a `WS_CLIPSIBLINGS` style:



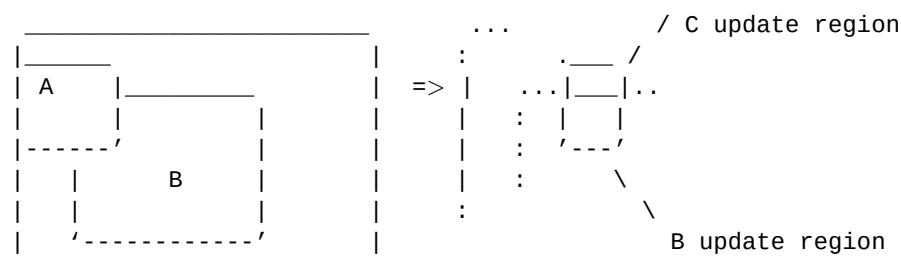
When the program requests a *display context* (DC) for a window it can specify an optional clipping region that further restricts the area where the graphics output can appear. This area is calculated as an intersection of the visible region and a clipping region.

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 will not receive another `WM_PAINT` until the window manager creates a new update region.

A was moved to the left:



```
|                  C    |
'-----------------------'
```

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).

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 system won't skip over it.

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`.

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.

Pros of Native DLLs

Native DLLs of course guarantee 100% compatibility for routines they implement. For example, using the native USER DLL would maintain a virtually perfect and Windows 95-like look for window borders, dialog controls, and 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.

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.

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.

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.

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 8. Debug Logging

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.

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.

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:
>
> ```
> dprintf_reg(stddeb, "Could not access key!\n");
> ```
>
> 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");
> ```

How to use it

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

```
#include "debugtools.h"

....

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

Some examples from the code:

```
#include "debugtools.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.

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.

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.

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 "debugtools.h"

...

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

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.
  --debugmsg yyy-all   -- disables debug messages for class yyy on all
                          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

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.

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)\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 additional information available, where in the second one, there is (e.g. the word stub)
- 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.

Chapter 9. COM/OLE in Wine

COM/OLE Architecture in Wine

The section goes into detail about how COM/OLE2 are implemented in Wine.

Using Binary OLE components in Wine

This section describes how to import pre-compiled COM/OLE components...

Writing OLE Components for Wine

Based on the comments in `wine/include/wine/obj_base.h`.

This section describes how to create your own natively compiled COM/OLE components.

Macros to define a COM interface

The goal of the following set of definitions is to provide a way to use the same header file definitions to provide both a C interface and a C++ object oriented interface to COM interfaces. The type of interface is selected automatically depending on the language but it is always possible to get the C interface in C++ by defining CINTERFACE.

It is based on the following assumptions:

- all COM interfaces derive from IUnknown, this should not be a problem.
- the header file only defines the interface, the actual fields are defined separately in the C file implementing the interface.

The natural approach to this problem would be to make sure we get a C++ class and virtual methods in C++ and a structure with a table of pointer to functions in C. Unfortunately the layout of the virtual table is compiler specific, the layout of g++ virtual tables is not the same as that of an egcs virtual table which is not the same as that generated by Visual C+. There are workarounds to make the virtual tables compatible via padding but unfortunately the one which is imposed to the WINE emulator by the Windows binaries, i.e. the Visual C++ one, is the most compact of all.

So the solution I finally adopted does not use virtual tables. Instead I use inline non virtual methods that dereference the method pointer themselves and perform the call.

Let’s take Direct3D as an example:

```
#define ICOM_INTERFACE IDirect3D
#define IDirect3D_METHODS \
    ICOM_METHOD1(HRESULT,Initialize,    REFIID,) \
    ICOM_METHOD2(HRESULT,EnumDevices,   LPD3DENUMDEVICESCALLBACK„ LPVOID,) \
    ICOM_METHOD2(HRESULT,CreateLight,   LPDIRECT3DLIGHT*„ IUnknown*,) \
    ICOM_METHOD2(HRESULT,CreateMaterial,LPDIRECT3DMATERIAL*„ IUnknown*,) \
    ICOM_METHOD2(HRESULT,CreateViewport,LPDIRECT3DVIEWPORT*„ IUnknown*,) \
    ICOM_METHOD2(HRESULT,FindDevice,    LPD3DFINDDEVICESEARCH„ LPD3DFINDDEVICERESULT,)
#define IDirect3D_IMETHODS \
    IUnknown_IMETHODS \
    IDirect3D_METHODS
ICOM_DEFINE(IDirect3D,IUnknown)
#undef ICOM_INTERFACE

#ifdef ICOM_CINTERFACE
// *** IUnknown methods *** //
```

```
#define IDirect3D_QueryInterface(p,a,b) ICOM_CALL2(QueryInterface,p,a,b)
#define IDirect3D_AddRef(p)             ICOM_CALL (AddRef,p)
#define IDirect3D_Release(p)            ICOM_CALL (Release,p)
// *** IDirect3D methods *** //
#define IDirect3D_Initialize(p,a)       ICOM_CALL1(Initialize,p,a)
#define IDirect3D_EnumDevices(p,a,b)    ICOM_CALL2(EnumDevice,p,a,b)
#define IDirect3D_CreateLight(p,a,b)    ICOM_CALL2(CreateLight,p,a,b)
#define IDirect3D_CreateMaterial(p,a,b) ICOM_CALL2(CreateMaterial,p,a,b)
#define IDirect3D_CreateViewport(p,a,b) ICOM_CALL2(CreateViewport,p,a,b)
#define IDirect3D_FindDevice(p,a,b)     ICOM_CALL2(FindDevice,p,a,b)
#endif
```

Comments:

The ICOM_INTERFACE macro is used in the ICOM_METHOD macros to define the type of the ’this’ pointer. Defining this macro here saves us the trouble of having to repeat the interface name everywhere. Note however that because of the way macros work, a macro like ICOM_METHOD1 cannot use ’ICOM_INTERFACE##_VTABLE’ because this would give ’ICOM_INTERFACE_VTABLE’ and not ’IDirect3D_VTABLE’.

ICOM_METHODS defines the methods specific to this interface. It is then aggregated with the inherited methods to form ICOM_IMETHODS.

ICOM_IMETHODS defines the list of methods that are inheritable from this interface. It must be written manually (rather than using a macro to generate the equivalent code) to avoid macro recursion (which compilers don’t like).

The ICOM_DEFINE finally declares all the structures necessary for the interface. We have to explicitly use the interface name for macro expansion reasons again. Inherited methods are inherited in C by using the IDirect3D_METHODS macro and the parent’s Xxx_IMETHODS macro. In C++ we need only use the IDirect3D_METHODS since method inheritance is taken care of by the language.

In C++ the ICOM_METHOD macros generate a function prototype and a call to a function pointer method. This means using once ’t1 p1, t2 p2, ...’ and once ’p1, p2’ without the types. The only way I found to handle this is to have one ICOM_METHOD macro per number of parameters and to have it take only the type information (with const if necessary) as parameters. The ’undef ICOM_INTERFACE’ is here to remind you that using ICOM_INTERFACE in the following macros will not work. This time it’s because the ICOM_CALL macro expansion is done only once the ’IDirect3D_Xxx’ macro is expanded. And by that time ICOM_INTERFACE will be long gone anyway.

You may have noticed the double commas after each parameter type. This allows you to put the name of that parameter which I think is good for documentation. It is not required and since I did not know what to put there for this example (I could only find doc about IDirect3D2), I left them blank.

Finally the set of ’IDirect3D_Xxx’ macros is a standard set of macros defined to ease access to the interface methods in C. Unfortunately I don’t see any way to avoid having to duplicate the inherited method definitions there. This time I could have used a trick to use only one macro whatever the number of parameters but I prefered to have it work the same way as above.

You probably have noticed that we don’t define the fields we need to actually implement this interface: reference count, pointer to other resources and miscellaneous fields. That’s because these interfaces are just that: interfaces. They may be implemented more than once, in different contexts and sometimes not even in Wine. Thus it would not make sense to impose that the interface contains some specific fields.

Bindings in C

In C this gives:

```
typedef struct IDirect3DVtbl IDirect3DVtbl;
struct IDirect3D {
    IDirect3DVtbl* lpVtbl;
};
```

```
struct IDirect3DVtbl {
    HRESULT (*fnQueryInterface)(IDirect3D* me, REFIID riid, LPVOID* ppvObj);
    ULONG (*fnAddRef)(IDirect3D* me);
    ULONG (*fnRelease)(IDirect3D* me);
    HRESULT (*fnInitialize)(IDirect3D* me, REFIID a);
    HRESULT (*fnEnumDevices)(IDirect3D* me, LPD3DENUMDEVICESCALLBACK a, LPVOID b);
    HRESULT (*fnCreateLight)(IDirect3D* me, LPDIRECT3DLIGHT* a, IUnknown* b);
    HRESULT (*fnCreateMaterial)(IDirect3D* me, LPDIRECT3DMATERIAL* a, IUnknown* b);
    HRESULT (*fnCreateViewport)(IDirect3D* me, LPDIRECT3DVIEWPORT* a, IUnknown* b);
    HRESULT (*fnFindDevice)(IDirect3D* me, LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESULT b);
};

#ifdef ICOM_CINTERFACE
// *** IUnknown methods *** //
#define IDirect3D_QueryInterface(p,a,b) (p)->lpVtbl->fnQueryInterface(p,a,b)
#define IDirect3D_AddRef(p)             (p)->lpVtbl->fnAddRef(p)
#define IDirect3D_Release(p)            (p)->lpVtbl->fnRelease(p)
// *** IDirect3D methods *** //
#define IDirect3D_Initialize(p,a)       (p)->lpVtbl->fnInitialize(p,a)
#define IDirect3D_EnumDevices(p,a,b)    (p)->lpVtbl->fnEnumDevice(p,a,b)
#define IDirect3D_CreateLight(p,a,b)    (p)->lpVtbl->fnCreateLight(p,a,b)
#define IDirect3D_CreateMaterial(p,a,b) (p)->lpVtbl->fnCreateMaterial(p,a,b)
#define IDirect3D_CreateViewport(p,a,b) (p)->lpVtbl->fnCreateViewport(p,a,b)
#define IDirect3D_FindDevice(p,a,b)     (p)->lpVtbl->fnFindDevice(p,a,b)
#endif
```

Comments:

IDirect3D only contains a pointer to the IDirect3D virtual/jump table. This is the only thing the user needs to know to use the interface. Of course the structure we will define to implement this interface will have more fields but the first one will match this pointer.

The code generated by ICOM_DEFINE defines both the structure representing the interface and the structure for the jump table. ICOM_DEFINE uses the parent's Xxx_IMETHODS macro to automatically repeat the prototypes of all the inherited methods and then uses IDirect3D_METHODS to define the IDirect3D methods.

Each method is declared as a pointer to function field in the jump table. The implementation will fill this jump table with appropriate values, probably using a static variable, and initialize the lpVtbl field to point to this variable.

The IDirect3D_Xxx macros then just derefence the lpVtbl pointer and use the function pointer corresponding to the macro name. This emulates the behavior of a virtual table and should be just as fast.

This C code should be quite compatible with the Windows headers both for code that uses COM interfaces and for code implementing a COM interface.

Bindings in C++

And in C++ (with gcc's g++):

```
typedef struct IDirect3D: public IUnknown {
    private: HRESULT (*fnInitialize)(IDirect3D* me, REFIID a);
    public: inline HRESULT Initialize(REFIID a) { return ((IDirect3D*)t.lpVtbl)->fnInitialize(this
    private: HRESULT (*fnEnumDevices)(IDirect3D* me, LPD3DENUMDEVICESCALLBACK a, LPVOID b);
    public: inline HRESULT EnumDevices(LPD3DENUMDEVICESCALLBACK a, LPVOID b)
        { return ((IDirect3D*)t.lpVtbl)->fnEnumDevices(this,a,b); };
    private: HRESULT (*fnCreateLight)(IDirect3D* me, LPDIRECT3DLIGHT* a, IUnknown* b);
    public: inline HRESULT CreateLight(LPDIRECT3DLIGHT* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateLight(this,a,b); };
    private: HRESULT (*fnCreateMaterial)(IDirect3D* me, LPDIRECT3DMATERIAL* a, IUnknown* b);
```

```
    public: inline HRESULT CreateMaterial(LPDIRECT3DMATERIAL* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateMaterial(this,a,b); };
    private: HRESULT (*fnCreateViewport)(IDirect3D* me, LPDIRECT3DVIEWPORT* a, IUnknown* b);
    public: inline HRESULT CreateViewport(LPDIRECT3DVIEWPORT* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateViewport(this,a,b); };
    private:  HRESULT (*fnFindDevice)(IDirect3D* me, LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESUL
    public: inline HRESULT FindDevice(LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESULT b)
        { return ((IDirect3D*)t.lpVtbl)->fnFindDevice(this,a,b); };
};
```

Comments:

In C++ IDirect3D does double duty as both the virtual/jump table and as the interface definition. The reason for this is to avoid having to duplicate the mehod definitions: once to have the function pointers in the jump table and once to have the methods in the interface class. Here one macro can generate both. This means though that the first pointer, t.lpVtbl defined in IUnknown, must be interpreted as the jump table pointer if we interpret the structure as the interface class, and as the function pointer to the QueryInterface method, t.fnQueryInterface, if we interpret the structure as the jump table. Fortunately this gymnastic is entirely taken care of in the header of IUnknown.

Of course in C++ we use inheritance so that we don't have to duplicate the method definitions.

Since IDirect3D does double duty, each ICOM_METHOD macro defines both a function pointer and a non-virtual inline method which dereferences it and calls it. This way this method behaves just like a virtual method but does not create a true C++ virtual table which would break the structure layout. If you look at the implementation of these methods you'll notice that they would not work for void functions. We have to return something and fortunately this seems to be what all the COM methods do (otherwise we would need another set of macros).

Note how the ICOM_METHOD generates both function prototypes mixing types and formal parameter names and the method invocation using only the formal parameter name. This is the reason why we need different macros to handle different numbers of parameters.

Finally there is no IDirect3D_Xxx macro. These are not needed in C++ unless the CINTERFACE macro is defined in which case we would not be here.

This C++ code works well for code that just uses COM interfaces. But it will not work with C++ code implement a COM interface. That's because such code assumes the interface methods are declared as virtual C++ methods which is not the case here.

Implementing a COM interface.

This continues the above example. This example assumes that the implementation is in C.

```
typedef struct _IDirect3D {
    void* lpVtbl;
    // ...
 } _IDirect3D;

static ICOM_VTABLE(IDirect3D) d3dvt;

// implement the IDirect3D methods here

int IDirect3D_fnQueryInterface(IDirect3D* me)
{
    ICOM_THIS(IDirect3D,me);
    // ...
}

// ...
```

```
static ICOM_VTABLE(IDirect3D) d3dvt = {
    ICOM_MSVTABLE_COMPAT_DummyRTTIVALUE
    IDirect3D_fnQueryInterface,
    IDirect3D_fnAdd,
    IDirect3D_fnAdd2,
    IDirect3D_fnInitialize,
    IDirect3D_fnSetWidth
};
```

Comments:

We first define what the interface really contains. This is the _IDirect3D structure. The first field must of course be the virtual table pointer. Everything else is free.

Then we predeclare our static virtual table variable, we will need its address in some methods to initialize the virtual table pointer of the returned interface objects.

Then we implement the interface methods. To match what has been declared in the header file they must take a pointer to a IDirect3D structure and we must cast it to an _IDirect3D so that we can manipulate the fields. This is performed by the ICOM_THIS macro.

Finally we initialize the virtual table.

Chapter 10. Wine and OpenGL

Written by Lionel Ulmer <lionel.ulmer@free.fr>, last modification : 2000/06/13

(Extracted from wine/documentation/opengl)

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.

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.

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*.

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.

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 :-).

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 `~/.wine/config` file in the [x11drv] section :

```
DesktopDoubleBuffered = Y
```

and to run Wine with the *`--desktop`* option.

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 :-)

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`.

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.

Known problems - shortcomings

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/)).

OpenGL not detected at configure time

See section (I) for a detailed explanation of the `configure` requirements.

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.

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.

`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 11. The Wine Build System

How the Wine build system works, and how to tweak it...

Chapter 12. Wine Builtin DLLs Overview

A more detailed look at Wine's builtin DLLs...

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)

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.

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.

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.

3. Controls

This section describes the development status of the common controls.

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??

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??

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??

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??

3.5 Flat Scroll Bar Control

Author:

Dummy written by Alex Priem <alex@sci.kun.nl>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

3.6 Header Control

Author:

Dummy written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

- Almost finished.
- Unicode notifications are not supported (WM_NOTIFYFORMAT).
- Order array not supported.

3.7 Hot Key Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

3.8 Image List (no control)

Author:

Dummy written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Almost finished.

3.9 IP Address Control

Author:

Dummy written by Eric Kohl <ekohl@abo.rhein-zeitung.de>, Alex Priem <alex@sci.kun.nl>

Status:

Under construction.

3.10 List View Control

Author:

Dummy written by:

- 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.

3.11 Month Calendar Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

3.12 Native font control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Dummy control. No functionality.

Notes:

Author needed!! Any volunteers??

3.13 Pager Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Under construction. Many missing features.

Notes:

Author needed!! Any volunteers??

3.14 Progress Bar Control

Author:

Original implementation by Dimitrie O. Paun. Fixes and improvements by Eric Kohl.

Status:

Finished!

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.

3.16 Rebar Control (Cool Bar)

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Development in progress. Many bugs and missing features.

Notes:

Author needed!! Any volunteers??

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.

3.18 Tab Control

Author:

Anders Carlsson <anders.carlsson@linux.nu>

Status:

Development in progress.

3.19 Toolbar Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Development in progress. Basic functionality is almost done. (dll version 4.0)

3.20 Tooltip Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de>

Status:

Almost finished.

Notes:

Unicode support is incomplete (WM_NOTIFYFORMAT).

3.21 Trackbar Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de> and Alex Priem <alex@sci.kun.nl>

Status:

Under construction.

3.22 Tree View Control

Author:

Written by Eric Kohl <ekohl@abo.rhein-zeitung.de> and Alex Priem <alex@sci.kun.nl>, fixes by Aric Stewart <aric@codeweavers.com>

Status:

Quite usable already.

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??

4. Additional Information

Has to be written...

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`.

5.1 Dynamic 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.

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.

5.3 "Most Recently Used" - List (MRU)

Only stubs are implemented to keep Explorer from bailing out.

No more information available at this time!

5.4 MenuHelp

Has to be written...

5.5 GetEffectiveClientRect

Has to be written...

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.

5.7 Other undocumented functions

Several other undocumented functions are used by IE4.

String functions: (will be written...)

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.

Chapter 13. Wine and Multimedia

This file contains information about the implementation of the multimedia layer of WINE.

The implementation can be found in the dlls/winmm/ directory (and in many of its subdirectories), but also in dlls/msacm/ (for the audio compression/decompression manager) and dlls/msvideo/ (for the video compression/decompression manager).

Written by Eric Pouech <Eric.Pouech@wanadoo.fr> (Last updated: 02/16/2001)

Overview

The multimedia stuff is split into 3 layers. The low level (device drivers), mid level (MCI commands) and high level abstraction layers. The low level layer has also some helper DLLs (like the MSACM/MSACM32 and MSVIDEO/MSVFW32 pairs).

The low level layer may depend on current hardware and OS services (like OSS on Unix). Mid level (MCI) and high level layers must be written independently from the hardware and OS services.

There are two specific low level drivers (one for wave input/output, another one for MIDI output only), whose role is:

- help choosing one low level driver between many
- add the possibility to convert streams (ie ADPCM => PCM)
- add the possibility to filter a stream (adding echo, equalizer... to a wave stream), or modify the instruments that have to be played (MIDI).

All of those components are defined as DLLs (one by one).

Low level layers

Please note that native low level drivers are not currently supported in WINE, because they either access hardware components or require VxDs to be loaded; WINE does not correctly supports those two so far.

The following low level layers are implemented (as built-in DLLs):

(Wave form) Audio

MMSYSTEM and WINMM call the real low level audio driver using the wodMessage/widMessage which handles the different requests.

OSS implementation

The low level audio driver is currently only implemented for the OpenSoundSystem (OSS) as supplied in the Linux and FreeBSD kernels by 4Front Technologies (http://www.4front-tech.com/). The presence of this driver is checked by configure (depends on the <sys/soundcard.h> file). Source code resides in dlls/winmm/wineoss/audio.c.

The implementation contains all features commonly used, but has several problems (see TODO list).

Note that some Wine specific flag has been added to the wodOpen function, so that the dsound DLL can share the /dev/dsp access. Currently, this only provides mutual exclusion for both DLLs. Future extension could add a virtual mixer between the two output streams.

TODO:

- verify all functions for correctness
- Add virtual mixer between wave-out and dsound interfaces.

Other sub systems

No other is available. Could think of Sun Audio, remote audio systems (using X extensions, ...), ALSA, EsounD, ArTs...

MIDI

MMSYSTEM and WINMM call the low level driver functions using the midMessage and the modMessage functions.

OSS driver

The low level audio driver is currently only implemented for the OpenSoundSystem (OSS) as supplied in the Linux and FreeBSD kernels by 4Front Technologies (http://www.4front-tech.com/). The presence of this driver is checked by configure (depends on the <sys/soundcard.h> file, and also some specific defines because MIDI is not supported on all OSes by OSS). Source code resides in dlls/winmm/wineoss/midi.c

Both Midi in and Midi out are provided. The type of MIDI devices supported is external MIDI port (requires an MIDI capable device - keyboard...) and OPL/2 synthesis (the OPL/2 patches for all instruments are in midiPatch.c).

TODO:

- use better instrument definition for OPL/2 (midiPatch.c) or use existing instrument definition (from playmidi or kmid) with a .winerc option
- have a look at OPL/3 ?
- implement asynchronous playback of MidiHdr
- implement STREAM'ed MidiHdr (question: how shall we share the code between the midiStream functions in MMSYSTEM/WINMM and the code for the low level driver)
- use a more accurate read mechanism than the one of snooping on timers (like select on fd)

Other sub systems

Could support other MIDI implementation for other sub systems (any idea here ?)

Could also implement a software synthesizer, either inside Wine or using using MIDI loop back devices in an external program (like timidity). The only trouble is that timidity is GPL'ed...

Mixer

MMSYSTEM and WINMM call the low level driver functions using the mixMessage function.

OSS implementation

The current implementation uses the OpenSoundSystem mixer, and resides in dlls/winmm/wineoss/mixer.c

TODO:

- implement notification mechanism when state of mixer's controls change

Other sub systems

TODO:

- implement mixing low level drivers for other mixers (ALSA...)

Aux

The AUX low level driver is the predecessor of the mixer driver (introduced in Win 95).

OSS driver

The implementation uses the OSS mixer API, and is incomplete.

TODO:

- verify the implementation
- check with what is done in mixer
- open question: shall we implement it on top of the low level mixer functions ?

Wine OSS

All the OSS dependent functions are stored into the WineOSS DLL. It still lack a correct installation scheme (as any multimedia device under Windows), so that all the correct keys are created in the registry. This requires an advanced model since, for example, the number of wave out devices can only be known on the destination system (depends on the sound card driven by the OSS interface). A solution would be to install all the multimedia drivers through the SETUPX DLL; this is not doable yet (the multimedia extension to SETUPX isn't written yet).

Joystick

The API consists of the joy* functions found in dlls/winmm/joystick/joystick.c. The implementation currently uses the Linux joystick device driver API. It is lacking support for enhanced joysticks and has not been extensively tested.

TODO:

- better support of enhanced joysticks (Linux 2.2 interface is available)
- support more joystick drivers (like the XInput extension)
- should load joystick DLL as any other driver (instead of hardcoding) the driver's name, and load it as any low lever driver.

Wave mapper (msacm.drv)

The Wave mapper device allows to load on-demand codecs in order to perform software conversion for the types the actual low level driver (hardware). Those codecs are provided through the standard ACM drivers.

Built-in

A first working implementation for wave out as been provided (wave in exists, but doesn't allow conversion).

Wave mapper driver implementation can be found in dlls/winmm/wavemap/ directory. This driver heavily relies on MSACM and MSACM32 DLLs which can be found in dlls/msacm and dlls/msacm32. Those DLLs load ACM drivers which provide the conversion to PCM format (which is normally supported by low level drivers). ADPCM, MP3... fit into the category of non PCM formats.

There is currently no built-in ACM driver in Wine, so you must use native ones if you're looking for non PCM playback.

TODO:

- check for correctness and robustness

Native

Seems to work quite ok (using of course native MSACM/MSACM32 DLLs) Some other testings report some issues while reading back the registry settings.

MIDI mapper

Midi mapper allows to map each one of 16 MIDI channels to a specific instrument on an installed sound card. This allows for example to support different MIDI instrument definition (XM, GM...). It also permits to output on a per channel basis to different MIDI renderers.

Built-in

A built-in MIDI mapper can be found in dlls/winmm/midimap/. It partly provides the same functionnality as the Windows' one. It allows to pick up destination channels (you can map a given channel to a specific playback device channel (see the configuration bits for more details).

TODO:

- implement the Midi mapper features (instrument on the fly modification) if it has to be done as under Windows, it required parsing the midi configuration files (didn't find yet the specs)

Native

The native midimapper from Win 98 works, but it requires a bunch of keys in the registry which are not part of the Wine source yet.

TODO:

- add native midimapper keys to the registry to let it run. This will require proper multimedia driver installation routines.

Mid level drivers (MCI)

The mid level drivers are represented by some common API functions, mostly mciSendCommand and mciSendString. See status in chapter 3 for more information. WINE implements several MCI mid level drivers (status is given for both built-in and native implementation):

TODO: (apply to all built-in MCI drivers)

- use MMSYSTEM multitasking caps instead of the home grown

CDAUDIO

Built-in

The currently best implementation is the MCI CDAUDIO driver that can be found in dlls/winmm/mcicda/mcicda.c. The implementation is mostly complete, there have been no reports of errors. It makes use of misc/cdrom.c Wine internal cdrom interface. This interface has been ported on Linux, FreeBSD and NetBSD. (Sun should be similar, but are not implemented.)

A very small example of a cdplayer consists just of the line mciSendString("play cdaudio",NULL,0,0);

TODO:

- add support for other cdaudio drivers (Solaris...)
- add support for multiple cdaudio devices (plus a decent configuration scheme)
- The DLL is not cleanly separated from the NTDLL DLL. The CDROM interface should be exported someway (or stored in a Wine only DLL)

Native

Native MCICDA works also correctly... It uses the MSCDEX traps (on int 2f). However, some commands (like seeking) seem to be broken.

MCIWAVE

Built-in

The implementation is rather complete and can be found in dlls/winmm/mciwave/audio.c. It uses the low level audio API (although not abstracted correctly).

FIXME:

- The MCI_STATUS command is broken.

TODO:

- check for correctness
- better use of asynchronous playback from low level
- better implement non waiting command (without the MCI_WAIT flag).

Native

Native MCIWAVE works also correctly.

MCISEQ (MIDI sequencer)

Built-in

The implementation can be found in dlls/winmm/mciseq/mcimidi.c. Except from the Record command, should be close to completion (except for non blocking commands, as many MCI drivers).

TODO:

- implement it correctly
- finish asynchronous commands (especially for reading/record)
- better implement non waiting command (without the MCI_WAIT flag).
- implement the recording features

Native

Native MCIMIDI has been working but is currently blocked by scheduling issues (mmTaskXXX no longer work).

FIXME:

- midiStreamPlay get from time to time an incorrect MidiHdr when using the native MCI sequencer

MCIANIM

Built-in

The implementation is in dlls/winmm/mcianim/.

TODO:

- implement it, probably using xanim or something similar.

Native

Native MCIANIM is reported to work (but requires native video DLLs also, even though the built-in video DLLs start to work correctly).

MCIAVI

Built-in

The implementation is in dlls/winmm/mcianim/. Basic features are present, simple playing is available, even if lots remain to be done. It rather heavily relies on MSVIDEO/MSVFW32 DLLs pair to work.

TODO:

- finish the implementation
- fix the audio/video synchronisation issue

Native

Native MCIAVI is reported to work (but requires native video DLLs also). Some files exhibit some deadlock issues anyway.

High level layers

The rest (basically the MMSYSTEM and WINMM DLLs entry points). It also provides the skeleton for the core functionality for multimedia rendering. Note that native MMSYSTEM and WINMM do not currently work under WINE and there is no plan to support them (it would require to also fully support VxD, which is not done yet). Moreover, native DLLs require 16 bit MCI and low level drivers. Wine implements them as 32 bit drivers. MCI and low level drivers can either be 16 or 32 bit for Wine.

TODO:

- it seems that some program check what's installed in registry against value returned by drivers. Wine is currently broken regarding this point.
- add clean-up mechanisms when process detaches from MM DLLs
- prepare for the 16/32 big split
- check thread-safeness for MMSYSTEM and WINMM entry points
- unicode entry points are badly supported

MCI skeleton

Implementation of what is needed to load/unload MCI drivers, and to pass correct information to them. This is implemented in dlls/winmm/mci.c. The mciSendString function uses command strings, which are translated into normal MCI commands as used by mciSendCommand with the help of command tables. The API can be found in dlls/winmm/mmsystem.c and dlls/winmm/mci.c. The functions there (mciOpen,mciSysInfo) handle mid level driver allocation and calls. The implementation is not complete.

MCI drivers are seen as regular WINE modules, and can be loaded (with a correct load order between built-in, native, elfdll, so), as any other DLL. Please note, that MCI drivers module names must bear the .drv extension to be correctly understood.

The list of available MCI drivers is obtained as follows: 1. key 'mci' in [option] section from .winerc (or wineconf) mci=CDAUDIO:SEQUENCER gives the list of MCI drivers (names, in uppercase only) to be used in WINE. 2. This list, when defined, supersedes the mci key in c:\windows\system.ini

Note that native VIDEODISC crashes when the module is loaded, which occurs when the MCI procedures are initialised. Make sure that this is not in the list from above. Try adding: mci=CDAUDIO:SEQUENCER:WAVEAUDIO:AVIVIDEO:MPEGVIDEO to the [options] section of wine.conf.

TODO:

- correctly handle the MCI_ALL_DEVICE_ID in functions.
- finish mapping 16 <=> 32 of MCI structures and commands
- MCI_SOUND is not handled correctly (should not be sent to MCI driver => same behavior as MCI_BREAK)
- implement auto-open feature (ie, when a string command is issued for a not yet opened device, MCI automatically opens it)

MCI multi-tasking

Multi-tasking capabilities used for the MCI drivers are provided in dlls/winmm/mmsystem.c.

TODO:

- mmTaskXXX functions are currently broken because the 16 loader does not support binary command lines => provide Wine's own mmtask.tsk not using binary command line.
- the Wine native MCI drivers should use the mmThreadXXX API (but since all built-in MCI drivers are 32 bit, this would require a special flag to mark 32 bit entry points)

Timers

It currently uses a service thread, run in the context of the calling process, which should correctly mimic Windows behavior.

TODO:

- Check if minimal time is satisfactory for most programs.
- current implementation may let a timer tick (once) after it has been destroyed

MMIO

The API consists of the mmio* functions found in mdlls/winmm/mmio.c. Seems to work ok in most of the cases. There's some linear/segmented issues with 16 bit code. There are also some bugs when writting MMIO files.

sndPlayXXX functions

Seem to work correctly.

Multimedia configuration

Currently, multimedia configuration heavily relies on Win 3.x configuration model.

Drivers

Since all multimedia drivers (MCI, low level ones, ACM drivers, mappers) are, at first, drivers they need to appear in the [mci] or [mci32] section of the system.ini file. Since all drivers are, at first, DLLs, you can choose to load their Wine's (built-in) or Windows (native) version.

MCI

A default [mci] section (in system.ini) looks like (see the note above on videodisc):

```
[mci]
cdaudio=mcicda.drv
sequencer=mciseq.drv
waveaudio=mciwave.drv
avivideo=mciavi.drv
videodisc=mcipionr.drv
vcr=mcivisca.drv
MPEGVideo=mciqtz.drv
```

By default, the list of loadable MCI drivers will be made of those drivers (in the [mci] section).

The list of loadable (recognized) MCI drivers can be altered in the [option] section of wine.conf, like: mci=CDAUDIO:SEQUENCER:WAVEAUDIO:AVIVIDEO:MPEGVIDEO

TODO:

- use a default registry setting to bypass this (ugly) configuration model

Low level drivers

Configuration of low level drivers is done with the Wine configuration file. Default keys are provided in winedefault.reg.

The registry keys used here differ from the Windows' one. Using the Windows' one would require implementing something equivalent to a (real) driver installation. Even if this would be necessary in a few cases (mainly using MS native multimedia) modules, there's no real need so far (or it hasn't been run into yet).

See the configuration part of the User's Guide for more details.

Midi mapper

The Midi mapper configuration is the same as on Windows 9x. Under the key

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Multimedia\MIDIMap
```

if the 'UseScheme' value is not set, or is set to a null value, the midi mapper will always use the driver identified by the 'CurrentInstrument' value. Note: Wine (for simplicity while installing) allows to define 'CurrentInstrument' as "#n" (where n is a number), whereas Windows only allows the real device name here. If UseScheme is set to a non null value, 'CurrentScheme' defines the name of the scheme to map the different channels. All the schemes are available with keys like

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\MediaProperties\PrivateProperties\Midi\Schemes
```

For every scheme, under this key, will be a sub-key (which name is usually a two digit index, starting at 00). Its default value is the name of the output driver, and the value 'Channels' lists all channels (of the 16 standard MIDI ones) which have to be copied to this driver.

To provide enhanced configuration and mapping capabilities, each driver can define under the key

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\MediaProperties\PrivateProperties\Midi\Ports\%`

a link to and .IDF file which allows to remap channels internally (for example 9 -> 16), to change instruments identification, event controlers values. See the source file dlls/winmm/midimap/midimap.c for the details (this isn't implemented yet).

ACM

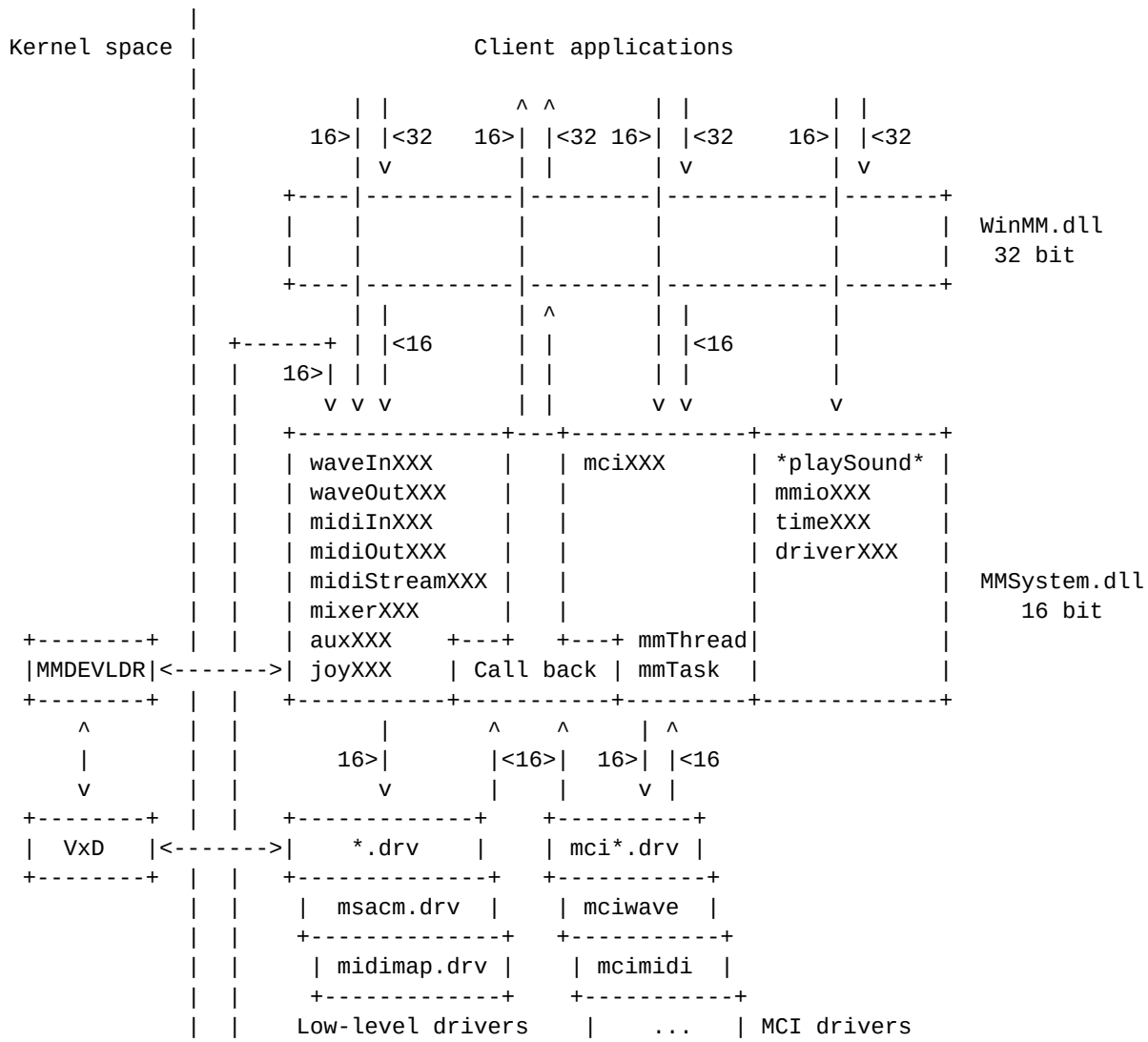
To be done (use the same mechanism as MCI drivers configuration).

VIDC

To be done (use the same mechanism as MCI drivers configuration).

Multimedia architecture

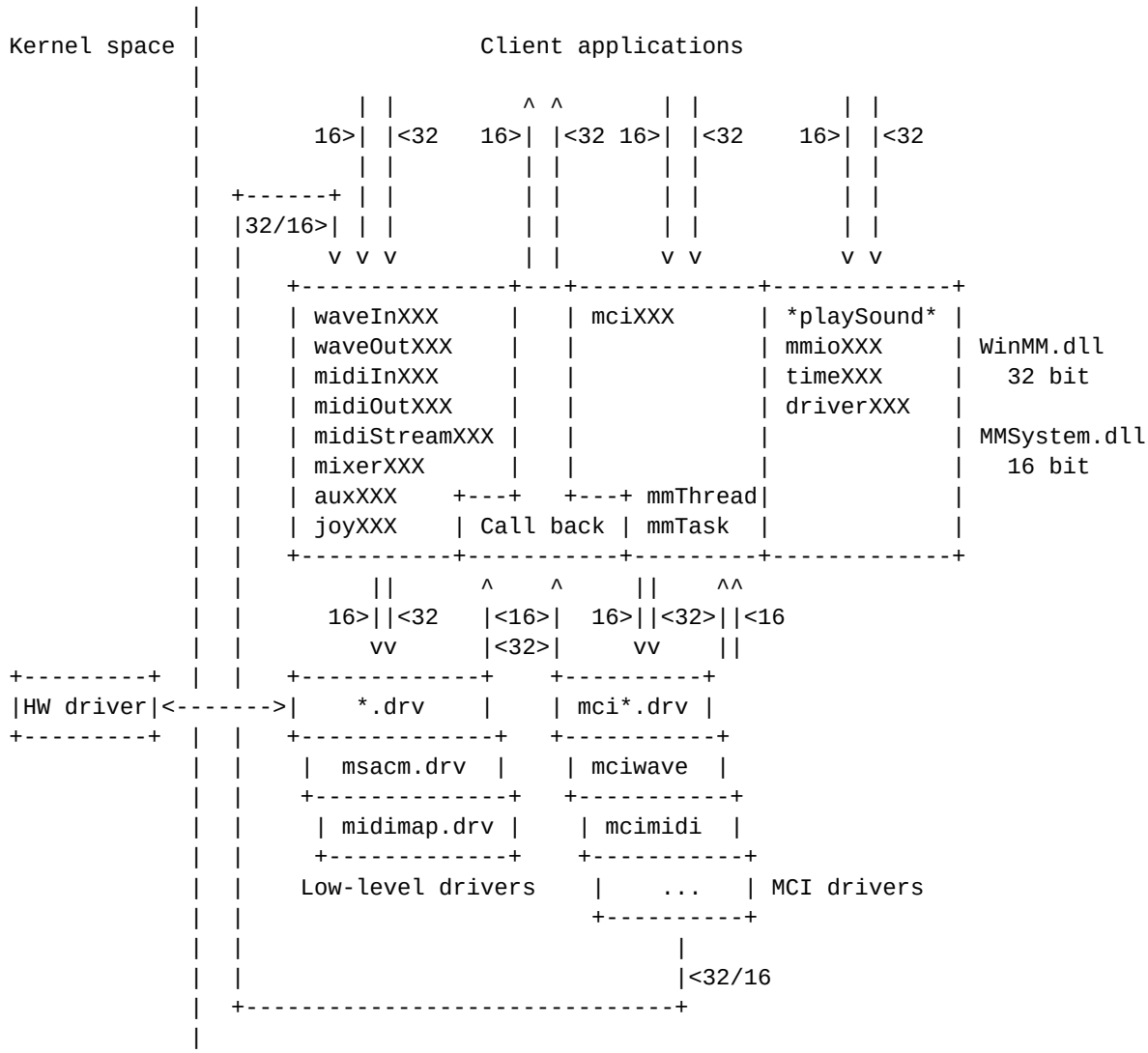
Windows 9x multimedia architecture



The important points to notice are:

- all drivers (and most of the core code) is 16 bit
- all hardware (or most of it) dependant code reside in the kernel space (which is not surprising)

Wine multimedia architecture



From the previous drawings, the most noticeable differences are:

- low-level drivers can either be 16 or 32 bit
- MCI drivers can either be 16 or 32 bit
- MMSystem and WinMM will be hosted in a single elfglue library

- no link between the MMSystem/WinMM pair on kernel space shall exist. For example, there will be a low level driver to talk to a UNIX OSS (Open Sound System) driver
- all built-in drivers (low-level and MCI) will be written as 32 bit drivers
- all native drivers will be 16 bits drivers

MS ACM Dlls

Contents

tbd

Status

tbd

Caching

The MSACM/MSACM32 keeps some data cached for all known ACM drivers. Under the key

```
Software\Microsoft\AudioCompressionManager\DriverCache\<driver
name>
```

are kept for values:

- aFormatTagCache which contains an array of DWORD. There are two DWORDs per cFormatTags entry. The first DWORD contains a format tag value, and the second the associated maximum size for a WAVEFORMATEX structure. (Fields dwFormatTag and cbFormatSize from ACMFORMATDETAILS)
- cFilterTags contains the number of tags supported by the driver for filtering.
- cFormatTags contains the number of tags support by the driver for conversions.
- fdwSupport (the same as the one returned from acmDriverDetails).

The cFilterTags, cFormatTags, fdwSupport are the same values as the ones returned from acmDriverDetails function.

III. Advanced Topics

Chapter 14. Low-level Implementation

Details of Wine's Low-level Implementation...

Builtin DLLs

Written by Juergen Schmied <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.

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)
  { ...
  }
  ....
}
```

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");
    exit (1);
  }
 }
....
```

3. Getting resources from a `*.rc` file linked to the DLL

< If you know how, write some lines>

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)`.

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 : \
 (handle)-5)

#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`).

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
```

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;
                        }

        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++;
```

```
}
      else
{
  if (count > 1)
    fprintf (stdout, "# Last line repeated %i times #\n", count);
  fprintf (stdout, "%s", buff);
  strcpy (lastline, buff);
    count = 1;
}
    }
}
EOF
```

Chapter 15. Porting Wine to new Platforms

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

Porting

written by ???

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

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?

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.

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**.

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**.

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)
- **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) *
- Implement mmap() and SysV IPC in EMX *
- 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...)

Chapter 16. Consoles in Wine

Consoles

Written by John Richardson <jrichard@zko.dec.com> Maintained by Joseph Pranevich <jpranevich@lycos.com>

(Extracted from wine/documentation/console)

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 is 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 inherited 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.

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 ECHOs 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 than the NT4 console implementation).

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.

DOS (Generic) Console Support

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.

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 appropriate 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.

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. 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 controlling the versioning system. It's a big file (> 40 meg) with a name like wine-cvsdirs-<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 -zxfcvs-dirs-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 wine-cvsdirs-xxx snapshot was generated.

Note also that it is possible to do all this with a direct Cvs connection, of course. The full cvs file method is less painful for the winehq cvs server and probably a bit faster if you don't have a very good net connection.

Note: If you use Cvs directly from the winehq.com server, do not forget to add to your `.cvsrc` file:

```
cvs -z 3
update -dPA
diff -u
```

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 EDT"
```

The date format is `YYYY-MM-DD HH:MM:SS`. Using the EDT date format ensure that you will be able to extract patches in a way that will be compatible with the wine-cvs archive :
http://www.winehq.com/hypermail/wine-cvs

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
```

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.

If you have lot of hard disk free space (a full compile takes currently 400 Mb), copy the oldest known working version before updating it, it will save time if you need to go back (it's better to make distclean before going back in time, so you have to make everything if you don't backup the older version)

When you have found the day where the problem happened, continue the search using the wine-cvs archive (sorted by date) and a more precise cvs update including hour, minute, second :

```
cvs -d $CVSROOT update -D "1999-06-01 15:17:25 EDT"
```

This will allow you to find easily the exact patch that did it.

6. If you find the patch that is the cause of the problem, you have almost won; report about it on comp.emulators.windows.wine or susbscribe to wine-devel and post it there. 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 :-)

Winelib User’s Guide

Winelib User's Guide

Table of Contents

Chapter 1. Winelib Introduction

What is Winelib?

Winelib is a development toolkit which allows you to compile your Windows applications on Unix.

Most of Winelib's code consists of the Win32 API implementation. Fortunately this part is 100 percent shared with Wine. The remainder consists of Windows compatible headers and tools like the resource compiler (and even these are used when compiling Wine).

Thanks to the above, Winelib supports most C and C++ 32bit source code, resource and message files, and can generate graphical or console applications as well as dynamic libraries.

What is not supported is 16bit source code as the types it depends on (especially segmented pointers) are not supported by Unix compilers. Also missing are some of the more exotic features of Microsoft's compiler like native COM support and structured exception handling. So you may need to perform some modifications in your code when recompiling your application with Winelib. This guide is here to help you in this task.

What you gain by recompiling your application with Winelib is the ability to make calls to Unix APIs, directly from your Windows source code. This allows for a better integration with the Unix environment than is allowed by runnning an unmodified Windows application running in Wine. Another benefit is that a Winelib application can relatively easily be recompiled on a non-Intel architecture and run there without the need for a slow software emulation of the processor.

System requirements

The requirements for Winelib are similar to those for Wine.

Basically if you can run Wine on your computer then you can run Winelib. But the converse is not true. You can also build Winelib and Winelib applications on platforms not supported by Wine, typically platforms with a non i386 processor. But this is still pretty much an uncharted territory. It would be more reasonable to first target one of the more mundane i386-based platforms first.

The main difference is that the compiler becomes much more important. It is highly recommended that you use gcc, g++, and the GNU binutils. The more recent your gcc compiler the better. For any serious amount of code you should not consider anything older than gcc 2.95.2. The latest gcc snapshots contain some useful bug fixes and much better support for anonymous structs and unions. This can help reduce the number of changes you have to do in your code but these are not stable releases of the compiler so you may not want to use them in production.

Getting Started

Winemaker introduction

So what is needed to compile a Windows application with Winelib? Well, it really depends on the complexity of your application but here are some issues that are shared by all applications:

- the case of your files may be bad. For example they could be in all caps: `HELLO.C`. It's not very nice to work with and probably not what you intended.
- then the case of the filenames in your include statements may be wrong: maybe they include 'Windows.h' instead of 'windows.h'.
- your include statements may use '\' instead of '/'. '\' is not recognized by Unix compilers while '/' is recognized in both environments.

- you will need to perform the usual Dos to Unix text file conversion otherwise you'll get in trouble when the compiler considers that your '\' is not at the end of the line since it is followed by a pesky carriage return.
- you will have to write new makefiles.

The best way to take care of all these issues is to use winemaker.

Winemaker is a perl script which is designed to help you bootstrap the conversion of your Windows projects to Winelib. In order to do this it will go analyze your code, fixing the issues listed above and generate autoconf-based Makefiles.

Let's suppose that Wine/Winelib has been installed in the `/usr/local/wine` directory, and that you are already in the top directory of your sources. Then converting your project to Winelib may be as simple as just running the three commands below:

```
$ winemaker --lower-uppercase .
$ ./configure --with-wine=/usr/local/wine
$ make
```

But of course things are not always that simple which is why we have this guide at all.

Step by step guide

Let's retrace the steps above in more details.

`Getting the source`

First if you can try to get the sources together with the executables/libraries that they build. In the current state of winemaker having these around can help it guess what it is that your project is trying to build. Later, when it is able to understand Visual C++ projects, and if you use them, this will no longer be necessary. Usually the executables and libraries are in a `Release` or `Debug` subdirectory of the directory where the sources are. So it's best if you can transfer the source files and either of these directories to Linux. Note that you don't need to transfer the `.obj`, `.pch`, `.sbr` and other files that also reside in these directories; especially as they tend to be quite big.

`cd <root_dir>`

Then go to the root directory where are your source files. Winemaker can deal with a whole directory hierarchy at once so you don't need to go into a leaf directory, quite the contrary. Winemaker will automatically generate makefiles in each directory where one is required, and will generate a global makefile so that you can rebuild all your executables and libraries with a single **make** command.

`Making the source writable`

Then make sure you have write access to your sources. It may sound obvious, but if you copied your source files from a CD-ROM or if they are in Source Safe on Windows, chances are that they will be read-only. But Winemaker needs write access so that it can fix them. You can arrange that by running **chmod -R u+w .**. Also you will want to make sure that you have a backup copy of your sources in case something went horribly wrong, or more likely, just for reference at a later point. If you use a version control system you're already covered.

If you have already modified your source files and you want to make sure that winemaker will not make further changes to them then you can use the --nosource-fix option to protect them.

`Running winemaker`

Then you'll run winemaker. Here are the options you will most likely want to use.

`--lower-uppercase`
`--lower-all`

These options specify how to deal with files, and directories, that have an 'incorrect' case. `--lower-uppercase` specifies they should only be renamed if their name is all uppercase. So files that have a mixed case, like 'Hello.c' would not be renamed. `--lower-all` will rename any file. If neither is specified then no file or directory will be renamed, almost. As you will see later winemaker may still have to rename some files.

`--nobackup`

Winemaker normally makes a backup of all the files in which it does more than the standard Dos to Unix conversion. But if you already have (handy) copies of these files elsewhere you may not need these so you should use this option.

`--dll`
`--console`

These option lets winemaker know what kind of target you are building. If you have the windows library in your source hierarchy then you should not need to specify `--dll`. But if you have console executables then you will need to use the corresponding option.

`--mfc`

This option tells winemaker that you are building an MFC application/library.

`-Dmacro[=defn]`
`-Idir`
`-Ldir`
`-idll`
`-llibrary`

The `-i` specifies a Winelib library to import via the spec file mechanism. Contrast this with the `-l` which specifies a Unix library to link with. The other options work the same way they would with a C compiler. All are applied to all the targets found. When specifying a directory with either `-I` or `-L`, winemaker will prefix a relative path with `$(TOPDIRECTORY)/` so that it is valid from any of the source directories. You can also use a variable in the path yourself if you wish (but don't forget to escape the '$'). For instance you could specify `-I\$(WINELIB_INCLUDE_ROOT)/msvcrt`.

So your command may finally look like: `winemaker --lower-uppercase -Imylib/include .`

`File renaming`

When you execute winemaker it will first rename files to bring their character case in line with your expectations and so that they can be processed by the makefiles. This later category implies that files with a non lowercase extension will be renamed so that the extension is in lowercase. So, for instance, `HELLO.C` will be renamed to `HELLO.c`. Also if a file or directory name contains a space or a dollar, then this character will be replaced with an underscore. This is because these characters cause problems with current versions of autoconf (2.13) and make (3.79).

`Source modifications and makefile generation`

winemaker will then proceed to modify the source files so that they will compile more readily with Winelib. As it does so it may print warnings when it has to make a guess or identifies a construct that it cannot correct. Finally it will generate the autoconf-based makefiles. Once all this is done you can review the changes that winemaker did to your files by using **diff -uw**. For instance: **diff -uw hello.c.bak hello.c**

`Running the configure script`

Before you run **make** you must run the autoconf **configure** script. The goal of this step is to analyze your system and generate customized makefiles from the `Makefile.in` files. This is also when you have to tell where Winelib resides on your system. If wine is installed in a single directory or you have the Wine sources compiled somewhere then you can just run **./configure --with-wine=/usr/local/bin** or **./configure --with-wine=~/wine** respectively.

`Running make`

This is a pretty simple step: just type **make** and voila, you should have all your executables and libraries. If this did not work out, then it means that you will have to read this guide further to:

- review the `Makefile.in` files to adjust the default compilation and link options set by winemaker. See the *Winemaker's source analysis* section for some hints.
- fix the portability issues in your sources. See *Portability issues* for more details.

If you find yourself modifying the Makefile.in to specify the location of the Wine header or library files then go back to the previous step (the configure script) and use the various --with-wine-* options to specify where they are.

Chapter 2. Portability issues

Anonymous unions/structs

Anonymous structs and unions support depends heavily on the compiler. The best support is provided by gcc/g++ 2.96 and later. But these versions of gcc come from the development branch so you may want to hold off before using them in production. g++ 2.95 supports anonymous unions but not anonymous structs and gcc 2.95 supports neither. Older versions of gcc/g++ have no support for either. since it is anonymous unions that are the most frequent in the windows headers, you should at least try to use gcc/g++ 2.95.

But you are stuck with a compiler that does not support anonymous structs/unions all is not lost. The Wine headers should detect this automatically and define `NONAMELESSUNION` / `NONAMELESSSTRUCT`. Then any anonymous union will be given a name `u` or `u2`, `u3`, etc. to avoid name clashes. You will then have to modify your code to include those names where appropriate.

The name that Wine adds to anonymous unions should match that used by the Windows headers. So all you have to do to compile your modified code in Windows is to explicitly define the `NONAMELESSUNION` macro. Note that it would be wise to also explicitly define this macro on in your Unix makefile (`Makefile.in`) to make sure your code will compile even if the compiler does support anonymous unions.

Things are not as nice when dealing with anonymous structs. Unfortunately the Windows headers make no provisions for compilers that do not support anonymous structs. So you will need to be more subtle when modifying your code if you still want it to compile in Windows. Here's a way to do it:

```
#ifdef WINELIB
#define ANONS .s
#else
#define ANONS
#endif

. . .

{
SYSTEM_INFO si;
GetSystemInfo(&si);
printf("Processor architecture=%d\n",si ANONS .wProcessorArchitecture);
}
```

You may put the `#define` directive directly in the source if only few files are impacted. Otherwise it's probably best to put it in one of your project's widely used headers. Fortunately usage of an anonymous struct is much rarer than usage of an anonymous union so these modifications should not be too much work.

Unicode

Because gcc and glibc use 4 byte unicode characters, the compiler intrinsic `L"foo"` generates unicode strings which cannot be used by Winelib (Win32 code expects 16 bit unicode characters). There are 3 workarounds for this:

1. Use the latest gcc version (2.9.7 or later), and pass the *`-fshort-wchar`* option to every file that is built.
2. Use the `__TEXT("foo")` macro, define `WINE_UNICODE_REWRITE` for each file that is built, and add *`-fwritable-strings`* to the compiler command line. You should replace all occurances of wchar_t with WCHAR also, since wchar_t is the native (32 bit) type. These changes allow Wine to modify the native unicode strings created by the compiler in place, so that they are 16 bit by the time any functions get to use them. This scheme works with older versions of gcc (2.95.x+).

3. Use the compiler default, but don't call any Win32 unicode function without converting the strings first!

If you are using Unicode and you want to be able to use standard library calls (e.g. `wcslen`, `wsprintf`) as well as Win32 unicode calls (API functions ending in W, or having `_UNICODE` defined), then you should use the msvcrt runtime library instead of glibc. The functions in glibc will not work correctly with 16 bit strings.

If you need a Unicode string even when `_UNICODE` isn't defined, use `WINE_UNICODE_TEXT("foo")`. This will need to be wrapped in `#ifdef WINELIB` to prevent breaking your source for windows compiles.

To prevent warnings when declaring a single unicode character in C, use `(WCHAR)L'x'`, rather than `__TEXT('x')`. This works on Windows also.

C library

There are 3 choices available to you regarding which C library to use:

1. Use the glibc native C library.
2. Use the msvcrt C library.
3. Use a custom mixture of both.

Note that under Wine, the crtdll library is implemented using msvcrt, so there is no benefit in trying to use it.

Using glibc in general has the lowest overhead, but this is really only important for file I/O. Many of the functions in msvcrt are simply resolved to glibc, so in reality options 2 and 3 are fairly similar choices.

To use glibc, you don't need to make changes to your application; it should work straight away. There are a few situations in which using glibc is not possible:

1. Your application uses Win32 and C library unicode functions.
2. Your application uses MS specific calls like `beginthread()`, `loadlibrary()`, etc.
3. You rely on the precise semantics of the calls, for example, returning `-1` rather than non-zero. More likely, your application will rely on calls like `fopen()` taking a Windows path rather than a Unix one.

In these cases you should use msvcrt to provide your C runtime calls. To do this, add a line:

```
import msvcrt.dll
```

to your applications `.spec` file. This will cause **winebuild** to resolve your c library calls to `msvcrt.dll`. Many simple calls which behave the same have been specified as non-importable from msvcrt; in these cases **winebuild** will not resolve them and the standard linker **ld** will link to the glibc version instead.

In order to avoid warnings in C (and potential errors in C++) from not having prototypes, you may need to use a set of MS compatable header files. These are scheduled for inclusion into Wine but at the time of writing are not available. Until they are, you can try prototyping the functions you need, or just live with the warnings.

If you have a set of include files (or when they are available in Wine), you need to use the *`-isystem "include_path"`* flag to gcc to tell it to use your headers in preference to the local system headers.

To use option 3, add the names of any symbols that you don't want to use from msvcrt into your applications `.spec` file. For example, if you wanted the MS specific functions, but not file I/O, you could have a list like:

```
@ignore = ( fopen fclose fwrite fread fputs fgets )
```

Obviously, the complete list would be much longer. Remember too that some functions are implemented with an underscore in their name and `#defined` to that name in the MS headers. So you may need to find out the name by examing `dlls/msvcrt/msvcrt.spec` to get the correct name for your `@ignore` entry.

Compiling Problems

If you get undefined references to Win32 API calls when building your application: if you have a VC++ `.dsp` file, check it for all the `.lib` files it imports, and add them to your applications `.spec` file. **winebuild** gives you a warning for unused imports so you can delete the ones you don't need later. Failing that, just import all the DLL's you can find in the `dlls/` directory of the Wine source tree.

If you are missing GUIDs at the link stage, add `-lwine_uuid` to the link line.

gcc is more strict than VC++, especially whan compiling C++. This may require you to add casts to your C++ to prevent overloading abiguities between similar types (such as two overloads that take int and char respectively).

If you come across a difference between the Windows headers and Wine's that breaks compilation, try asking for help on <`wine-devel@winehq.com`>.

Initialization problems

Initialization problems occur when the application calls the Win32 API before Winelib has been initialized. How can this happen?

Winelib is initialized by the application's `main` before it calls the regular `WinMain`. But, in C++, the constructors of static class variables are called before the `main` (by the module's initializer). So if such a constructor makes calls to the Win32 API, Winelib will not be initialized at the time of the call and you may get a crash. This problem is much more frequent in C++ because of these class constructors but could also, at least in theory, happen in C if you were to specify an initializer making calls to Winelib. But of course, now that you are aware of this problem you won't do it :-).

Further compounding the problem is the fact that Linux's (GNU's?) current dynamic library loader does not call the module initializers in their dependency order. So even if Winelib were to have its own initializer there would be no garantee that it would be called before the initializer of the library containing this static variable. Finally even if the variable is in a library that your application links with, that library's initializer may be called before Winelib has been initialized. One such library is the MFC.

The current workaround is to move all the application's code in a library and to use a small Winelib application to dynamically load this library. Tus the initialization sequence becomes:

- the wrapper application starts.
- its empty initializer is run.
- its `main` is run. Its first task is to initialize Winelib.
- it then loads the application's main library, plus all its dependent libraries.
- which triggers the execution of all these libraries initializers in some unknown order. But all is fine because Winelib has already been initialized anyway.
- finally the main function calls the `WinMain` of the application's library.

This may sound complex by Winemaker makes it simple. Just specify `--wrap` or `--mfc` on the command line and it will adapt its makefiles to build the wrapper and the application library.

VC's native COM support

don't use it, guide on how to replace it with normal C++ code (yes, how???): extracting a .h and .lib from a COM dll Can '-fno-rtti' be of some use or even required?

SEH

how to modify the syntax so that it works both with gcc's macros and Wine's macros, is it even possible?

Others

-fpermissive and -fno-for-scope, maybe other options

Chapter 3. The Winelib development toolkit

Winemaker

Support for Visual C++ projects

Unfortunately Winemaker does not support the Visual C++ project files, ...yet. Supporting Visual C++ project files (the `.dsp` and some `.mak` files for older versions of Visual C++) is definitely on the list of important Winemaker improvements as it will allow it to properly detect the defines to be used, any custom include path, the list of libraries to link with, and exactly which source files to use to build a specific target. All things that the current version of Winemaker has to guess or that you have to tell it as will become clear in the next section.

When the time comes Winemaker, and its associated build system, will need some extensions to support:

- per file defines and include paths. Visual C++ projects allow the user to specify compiler options for each individual file being compiled. But this is probably not very frequent so it might not be that important.
- multiple configurations. Visual C++ projects usually have at least a 'Debug' and a 'Release' configuration which are compiled with different compiler options. How exactly we deal with these configurations remains to be determined.

Winemaker's source analysis

Winemaker can do its work even without a Windows makefile or a Visual Studio project to start from (it would not know what to do with a windows makefile anyway). This involves doing many educated guesses which may be wrong. But by and large it works. The purpose of this section is to describe in more details how winemaker proceeds so that you can better understand why it gets things wrong and how to fix it/avoid it.

At the core winemaker does a recursive traversal of your source tree looking for targets (things to build) and source files. Let's start with the targets.

First are executables and dlls. Each time it finds one of these in a directory, winemaker puts it in the list of things to build and will later generate a `Makefile.in` file in this directory. Note that Winemaker also knows about the commonly used `Release` and `Debug` directories, so it will attribute the executables and libraries found in these to their parent directory. When it finds an executable or a dll winemaker is happy because these give it more information than the other cases described below.

If it does not find any executable or dll winemaker will look for files with a `.mak` extension. If they are not disguised Visual C++ projects (and currently even if they are), winemaker will assume that a target by that name should be built in this directory. But it will not know whether this target is an executable or a library. So it will assume it is of the default type, i.e. a graphical application, which you can override by using the `--cuiexe` and `--dll` options.

Finally winemaker will check to see if there is a file called `makefile`. If there is, then it will assume that there is exactly one target to build for this directory. But it will not know the name or type of this target. For the type it will do as in the above case. And for the name it will use the directory's name. Actually, if the directory starts with `src` winemaker will try to make use of the name of the parent directory instead.

Once the target list for a directory has been established, winemaker will check whether it contains a mix of executables and libraries. If it is so, then winemaker will make it so that each executable is linked with all the libraries of that directory.

If the previous two steps don't produce the expected results (or you think they will not) then you should put winemaker in interactive mode (see *The interactive mode*). This will allow you to specify the target list (and more) for each directory.

In each directory winemaker also looks for source files: C, C++ or resource files. If it also found targets to build in this directory it will then try to assign each source file to one of these targets based on their names. Source files that do not seem to match any specific target are put in a global list for this directory, see the `EXTRA_xxx` variables in the `Makefile.in`, and linked with each of the targets. The assumption here is that these source files contain common code which is shared by all the targets. If no targets were found in the directory where these files are located, then they are assigned to the parent's directory. So if a target is found in the parent directory it will also 'inherit' the source files found in its subdirectories.

Finally winemaker also looks for more exotic files like `.h` headers, `.inl` files containing inline functions and a few others. These are not put in the regular source file lists since they are not compiled directly. But winemaker will still remember them so that they are processed when the time comes to fix the source files.

Fixing the source files is done as soon as winemaker has finished its recursive directory traversal. The two main tasks in this step are fixing the CRLF issues and verifying the case of the include statements.

Winemaker makes a backup of each source file (in such a way that symbolic links are preserved), then reads it fixing the CRLF issues and the other issues as it goes. Once it has finished working on a file it checks whether it has done any non CRLF-related modification and deletes the backup file if it did not (or if you used `--nobackup`).

Checking the case of the include statements (of any form, including files referenced by resource files), is done in the context of that source file's project. This way winemaker can use the proper include path when looking for the file that is included. If winemaker fails to find a file in any of the directories of the include path, it will rename it to lowercase on the basis that it is most likely a system header and that all system headers names are lowercase (this can be overriden by using `--nolower-include`).

Finally winemaker generates the `Makefile.in` files and other support files (wrapper files, spec files, `configure.in`, `Make.rules.in`). From the above description you can guess at the items that winemaker may get wrong in this phase: macro definitions, include path, library path, list of libraries to import. You can deal with these issues by using winemaker's `-D`, `-I`, `-L` and `-i` options if they are homogeneous enough between all your targets. Otherwise you may want to use winemaker's interactive mode so that you can specify different settings for each project / target.

For instance, one of the problems you are likely to encounter is that of the `STRICT` macro. Some programs will not compile if `STRICT` is not turned on, and others will not compile if it is. Fortunately all the files in a given source tree use the same setting so that all you have to do is add `-DSTRICT` on winemaker's command line or in the `Makefile.in` file(s).

Finally the most likely reasons for missing or duplicate symbols are:

- The target is not being linked with the right set of libraries. You can avoid this by using winemaker's `-L` and `-i` options or adding these libraries to the `Makefile.in` file.
- Maybe you have multiple targets in a single directory and winemaker guessed wrong when trying to match the source files with the targets. The only way to fix this kind of problem is to edit the `Makefile.in` file manually.
- Winemaker assumes you have organized your source files hierarchically. If a target uses source files that are in a sibling directory, e.g. if you link with `../hello/world.o` then you will get missing symbols. Again the only solution is to manually edit the `Makefile.in` file.

The interactive mode

what is it, when to use it, how to use it

The Makefile.in files

The `Makefile.in` is your makefile. More precisely it is the template from which the actual makefile will be generated by the `configure` script. It also relies on the `Make.rules` file for most of the actual logic. This way it

only contains a relatively simple description of what needs to be built, not the complex logic of how things are actually built.

So this is the file to modify if you want to customize things. Here's a detailed description of its content:

```
### Generic autoconf variables

TOPSRCDIR             = @top_srcdir@
TOPOBJDIR             = .
SRCDIR                = @srcdir@
VPATH                 = @srcdir@
```

The above is part of the standard autoconf boiler-plate. These variables make it possible to have per-architecture directories for compiled files and other similar goodies (But note that this kind of functionality has not been tested with winemaker generated `Makefile.in` files yet).

```
SUBDIRS               =
DLLS                  =
EXES                  = hello
```

This is where the targets for this directory are listed. The names are pretty self-explanatory. `SUBDIRS` is usually only present in the top-level makefile. For libraries you should put the full Unix name, e.g. `libfoo.so`.

```
### Global settings

DEFINES               = -DSTRICT
INCLUDE_PATH          =
LIBRARY_PATH          =
LIBRARIES             =
```

This section contains the global compilation settings: they apply to all the targets in this makefile. The `LIBRARIES` variable allows you to specify additional Unix libraries to link with. Note that you would normally not specify Winelib libraries there. To link with a Winelib library, one uses the 'import' statement of the spec files. The exception is when you have not explicitly exported the functions of a Winelib library. One library you are likely to find here is `mfc` (note, the '-l' is omitted).

The other variable names should be self-explanatory. You can also use three additional variables that are usually not present in the file: `CEXTRA`, `CXXEXTRA` and `WRCEXTRA` which allow you to specify additional flags for, respectively, the C compiler, the C++ compiler and the resource compiler. Finally note that all these variable contain the option's name except `IMPORTS`. So you should put `-DSTRICT` in `DEFINES` but `winmm` in `IMPORTS`.

Then come one section per target, each describing the various components that target is made of.

```
### hello sources and settings

hello_C_SRCS          = hello.c
hello_CXX_SRCS        =
hello_RC_SRCS         =
hello_SPEC_SRCS       = hello.spec
```

Each section will start with a comment indicating the name of the target. Then come a series of variables prefixed with the name of that target. Note that the name of the prefix may be slightly different from that of the target because of restrictions on the variable names.

The above variables list the sources that are used togenerate the target. Note that there should only be one resource file in `RC_SRCS`, and that `SPEC_SRCS` will always contain a single spec file.

```
hello_LIBRARY_PATH    =
```

```
hello_LIBRARIES       =
hello_DEPENDS         =
```

The above variables specify how to link the target. Note that they add to the global settings we saw at the beginning of this file.

`DEPENDS`, when present, specifies a list of other targets that this target depends on. Winemaker will automatically fill this field, and the `LIBRARIES` field, when an executable and a library are built in the same directory.

The reason why winemaker also links with libraries in the Unix sense in the case above is because functions will not be properly exported. Once you have exported all the functions in the library's spec file you should remove them from the `LIBRARIES` field.

```
hello_OBJS            = $(hello_C_SRCS:.c=.o) \
                        $(hello_CXX_SRCS:.cpp=.o) \
                        $(EXTRA_OBJS)
```

The above just builds a list of all the object files that correspond to this target. This list is later used for the link command.

```
### Global source lists

C_SRCS                = $(hello_C_SRCS)
CXX_SRCS              = $(hello_CXX_SRCS)
RC_SRCS               = $(hello_RC_SRCS)
SPEC_SRCS             = $(hello_SPEC_SRCS)
```

This section builds 'summary' lists of source files. These lists are used by the `Make.rules` file.

```
### Generic autoconf targets

all: $(DLLS) $(EXES:%=%.so)

@MAKE_RULES@

install::
        for i in $(EXES); do $(INSTALL_PROGRAM) $$i $(bindir); done
        for i in $(EXES:%=%.so) $(DLLS); do $(INSTALL_LIBRARY) $$i $(libdir); done

uninstall::
        for i in $(EXES); do $(RM) $(bindir)/$$i;done
        for i in $(EXES:%=%.so) $(DLLS); do $(RM) $(libdir)/$$i;done
```

The above first defines the default target for this makefile. Here it consists in trying to build all the targets. Then it includes the `Make.rules` file which contains the build logic, and provides a few more standard targets to install / uninstall the targets.

```
### Target specific build rules

$(hello_SPEC_SRCS:.spec=.tmp.o): $(hello_OBJS)
        $(LDCOMBINE) $(hello_OBJS) -o $@
        -$(STRIP) $(STRIPFLAGS) $@

$(hello_SPEC_SRCS:.spec=.spec.c): $(hello_SPEC_SRCS:.spec) $(hello_SPEC_SRCS:.spec=.tmp.o) $(hello
        $(WINEBUILD) -fPIC $(hello_LIBRARY_PATH) $(WINE_LIBRARY_PATH) -sym $(hello_SPEC_SRCS:.spec

hello.so: $(hello_SPEC_SRCS:.spec=.spec.o) $(hello_OBJS) $(hello_DEP
```

```
ENDS)
        $(LDSHARED) $(LDDLLFLAGS) -o $@ $(hello_OBJS) $(hello_SPEC_SRCS:.spec=.spec.o) $(hello_LIB
        test -f hello || $(LN_S) $(WINE) hello
```

Then come additional directives to link the executables and libraries. These are pretty much standard and you should not need to modify them.

The Make.rules.in file

What's in the Make.rules.in...

The configure.in file

What's in the configure.in...

Compiling resource files: WRC

To compile resources you should use the Wine Resource Compiler, wrc for short, which produces a binary `.res` file. This resource file is then used by winebuild when compiling the spec file (see *The Spec file*).

Again the makefiles generated by winemaker take care of this for you. But if you were to write your own makefile you would put something like the following:

```
WRC=$(WINE_DIR)/tools/wrc/wrc

WINELIB_FLAGS = -I$(WINE_DIR)/include -DWINELIB -D_REENTRANT
WRCFLAGS      = -r -L

.SUFFIXES: .rc .res

.rc.res:
$(WRC) $(WRCFLAGS) $(WINELIB_FLAGS) -o $@ $<
```

There are two issues you are likely to encounter with resource files.

The first problem is with the C library headers. WRC does not know where these headers are located. So if an RC file, of a file it includes, references such a header you will get a 'file not found' error from wrc. Here are a few ways to deal with this:

- The solution traditionally used by the Winelib headers is to enclose the offending include statement in an `#ifndef RC_INVOKED` statement where `RC_INVOKED` is a macro name which is automatically defined by wrc.
- Alternately you can add one or more `-I` directive to your wrc command so that it finds you system files. For instance you may add `-I/usr/include -I/usr/lib/gcc-lib/i386-linux/2.95.2/include` to cater to both C and C++ headers. But this supposes that you know where these header files reside which decreases the portability of your makefiles to other platforms (unless you automatically detect all the necessary directories in the autoconf script).

  Or you could use the C/C++ compiler to perform the preprocessing. To do so, simply modify your makefile as follows:

  ```
  .rc.res:
  $(CC) $(CC_OPTS) -DRC_INVOKED -E -x c $< | $(WRC) -N $(WRCFLAGS) $(WINELIB_FLAGS) -o $@
  ```

The second problem is that the headers may contain constructs that WRC fails to understand. A typical example is a function which return a 'const' type. WRC expects a function to be two identifiers followed by an opening parenthesis. With the const this is three identifiers followed by a parenthesis and thus WRC is confused (note: WRC should in fact ignore all this like the windows resource compiler does). The current work-around is to enclose offending statement(s) in an `#ifndef RC_INVOKED`.

Using GIF files in resources is problematic. For best results, convert them to BMP and change your `.res` file.

If you use common controls/dialogs in your resource files, you will need to add `#include <commctrl.h>` after the `#include <windows.h>` line, so that **wrc** knows the values of control specific flags.

Compiling message files: WMC

how does one use it???

The Spec file

Introduction

In Windows the program's life starts either when its `main` is called, for console applications, or when its `WinMain` is called, for windows applications in the 'windows' subsystem. On Unix it is always `main` that is called. Furthermore in Winelib it has some special tasks to accomplish, such as initializing Winelib, that a normal `main` does not have to do.

Furthermore windows applications and libraries contain some information which are necessary to make APIs such as `GetProcAddress` work. So it is necessary to duplicate these data structures in the Unix world to make these same APIs work with Winelib applications and libraries.

The spec file is there to solve the semantic gap described above. It provides the `main` function that initializes Winelib and calls the module's `WinMain` / `DllMain`, and it contains information about each API exported from a Dll so that the appropriate tables can be generated.

A typical spec file will look something like this:

```
name    hello
type    win32
mode    guiexe
init    WinMain
rsrc    resource.res

import  winmm.dll
```

And here are the entries you will probably want to change:

name

This is the name of the Win32 module. Usually this is the same as that of the application or library (but without the 'lib' and the '.so').

mode
init

`mode` defines whether what you are building is a library, `dll`, a console application, `cuiexe` or a regular graphical application `guiexe`. Then `init` defines what is the entry point of that module. For a library this is

customarily set to `DllMain`, for a console application this is `main` and for a graphical application this is `WinMain`.

import

Add an 'import' statement for each library that this executable depends on. If you don't, these libraries will not get initialized in which case they may very well not work (e.g. winmm).

rsrc

This item specifies the name of the compiled resource file to link with your module. If your resource file is called `hello.rc` then the wrc compilation step (see *Compiling resource files: WRC*) will generate a file called `hello.res`. This is the name you must provide here. Note that because of this you cannot compile the spec file before you have compiled the resource file. So you should put a rule like the following in your makefile:

```
hello.spec.c: hello.res
```

If your project does not have a resource file then you must omit this entry altogether.

@

This entry is not shown above because it is not always necessary. In fact it is only necessary to export functions when you plan to dynamically load the library with `LoadLibrary` and then do a `GetProcAddress` on these functions. This is not necessary if you just plan on linking with the library and calling the functions normally. For more details about this see: *More details*.

Compiling it

Compiling a spec file is a two step process. It is first converted into a C file by winebuild, and then compiled into an object file using your regular C compiler. This is all taken care of by the winemaker generated makefiles of course. But here's what it would like if you had to do it by hand:

```
WINEBUILD=$(WINE_DIR)/tools/winebuild

.SUFFIXES: .spec .spec.c .spec.o

.spec.spec.c:
$(WINEBUILD) -fPIC -o $@ -spec $<

.spec.c.spec.o:
$(CC) -c -o $*.spec.o $<
```

Nothing really complex there. Just don't forget the `.SUFFIXES` statement, and beware of the tab if you copy this straight to your Makefile.

More details

(Extracted from tools/winebuild/README)

Here is a more detailed description of the spec file's format.

```
# comment text
```

Anything after a '#' will be ignored as comments.

```
name    NAME
type    win16|win32 <--- the |'s mean it's one or the other
```

These two fields are mandatory. `name` defines the name of your module and `type` whether it is a Win16 or Win32 module. Note that for Winelib you should only be using Win32 modules.

```
file    WINFILENAME
```

This field is optional. It gives the name of the Windows file that is replaced by the builtin. <`name`>`.DLL` is assumed if none is given. This is important for kernel, which lives in the Windows file `KRNL386.EXE`.

```
heap    SIZE
```

This field is optional and specific to Win16 modules. It defines the size of the module local heap. The default is no local heap.

```
mode    dll|cuiexe|guiexe
```

This field is optional. It specifies specifies whether it is the spec file for a dll or the main exe. This is only valid for Win32 spec files.

```
init    FUNCTION
```

This field is optional and specific to Win32 modules. It specifies a function which will be called when the dll is loaded or the executable started.

```
import  DLL
```

This field can be present zero or more times. Each instance names a dll that this module depends on (only for Win32 modules at the present).

```
rsrc    RES_FILE
```

This field is optional. If present it specifies the name of the .res file containing the compiled resources. See *Compiling resource files: WRC* for details on compiling a resource file.

```
ORDINAL VARTYPE EXPORTNAME (DATA [DATA [DATA [...]]])
2 byte Variable(-1 0xff 0 0)
```

This field can be present zero or more times. Each instance defines data storage at the ordinal specified. You may store items as bytes, 16-bit words, or 32-bit words. `ORDINAL` is replaced by the ordinal number corresponding to the variable. `VARTYPE` should be `byte`, `word` or `long` for 8, 16, or 32 bits respectively. `EXPORTNAME` will be the name available for dynamic linking. `DATA` can be a decimal number or a hex number preceeded by "0x". The example defines the variable `Variable` at ordinal 2 and containing 4 bytes.

```
ORDINAL equate EXPORTNAME DATA
```

This field can be present zero or more times. Each instance defines an ordinal as an absolute value. `ORDINAL` is replaced by the ordinal number corresponding to the variable. `EXPORTNAME` will be the name available for dynamic linking. `DATA` can be a decimal number or a hex number preceeded by "0x".

```
ORDINAL FUNCTYPE EXPORTNAME([ARGTYPE [ARGTYPE [...]]]) HANDLERNAME
100 pascal CreateWindow(ptr ptr long s_word s_word s_word s_word
                        word word word ptr)
           WIN_CreateWindow
101 pascal GetFocus() WIN_GetFocus()
```

This field can be present zero or more times. Each instance defines a function entry point. The prototype defined by `EXPORTNAME ([ARGTYPE [ARGTYPE [...]]])` specifies the name available for dynamic linking and the format

of the arguments. `"ORDINAL"` is replaced by the ordinal number corresponding to the function, or `@` for automatic ordinal allocation (Win32 only).

`FUNCTYPE` should be one of:

pascal16
: for a Win16 function returning a 16-bit value

pascal
: for a Win16 function returning a 32-bit value

register
: for a function using CPU register to pass arguments

interrupt
: for a Win16 interrupt handler routine

stdcall
: for a normal Win32 function

cdecl
: for a Win32 function using the C calling convention

varargs
: for a Win32 function taking a variable number of arguments

`ARGTYPE` should be one of:

word
: for a 16 bit word

long
: a 32 bit value

ptr
: for a linear pointer

str
: for a linear pointer to a null-terminated string

s_word
: for a 16 bit signed word

segptr
: for a segmented pointer

segstr
: for a segmented pointer to a null-terminated string

Only `ptr`, `str` and `long` are valid for Win32 functions. `HANDLERNAME` is the name of the actual Wine function that will process the request in 32-bit mode.

The two examples define an entry point for the `CreateWindow` and `GetFocus` calls respectively. The ordinals used are just examples.

To declare a function using a variable number of arguments in Win16, specify the function as taking no arguments. The arguments are then available with CURRENT_STACK16->args. In Win32, specify the function as `varargs` and

declare it with a '...' parameter in the C file. See the wsprintf* functions in `user.spec` and `user32.spec` for an example.

```
ORDINAL stub EXPORTNAME
```

This field can be present zero or more times. Each instance defines a stub function. It makes the ordinal available for dynamic linking, but will terminate execution with an error message if the function is ever called.

```
ORDINAL extern EXPORTNAME SYMBOLNAME
```

This field can be present zero or more times. Each instance defines an entry that simply maps to a Wine symbol (variable or function); `EXPORTNAME` will point to the symbol `SYMBOLNAME` that must be defined in C code. This type only works with Win32.

```
ORDINAL forward EXPORTNAME SYMBOLNAME
```

This field can be present zero or more times. Each instance defines an entry that is forwarded to another entry point (kind of a symbolic link). `EXPORTNAME` will forward to the entry point `SYMBOLNAME` that must be of the form `DLL.Function`. This type only works with Win32.

Linking it all together

To link an executable you need to link together: your object files, the spec file, any Windows libraries that your application depends on, gdi32 for instance, and any additional library that you use. All the libraries you link with should be available as '.so' libraries. If one of them is available only in '.dll' form then consult *Dealing with binary only dlls*.

It is also when attempting to link your executable that you will discover whether you have missing symbols or not in your custom libraries. On Windows when you build a library, the linker will immediately tell you if a symbol it is supposed to export is undefined. In Unix, and in Winelib, this is not the case. The symbol will silently be marked as undefined and it is only when you try to produce an executable that the linker will verify all the symbols are accounted for.

So before declaring victory when first converting a library to Winelib, you should first try to link it to an executable (but you would have done that to test it anyway, right?). At this point you may discover some undefined symbols that you thought were implemented by the library. Then, you to the library sources and fix it. But you may also discover that the missing symbols are defined in, say, gdi32. This is because you did not link the said library with gdi32. One way to fix it is to link this executable, and any other that also uses your library, with gdi32. But it is better to go back to your library's makefile and explicitly link it with gdi32.

As you will quickly notice, this has unfortunately not been (completely) done for Winelib's own libraries. So if an application must link with ole32, you will also need to link with advapi32, rpcrt4 and others even if you don't use them directly. This can be annoying and hopefully will be fixed soon (feel free to submit a patch).

Chapter 4. Dealing with the MFC

Introduction

To use the MFC in a Winelib application you will first have to recompile the MFC with Winelib. In theory it should be possible to write a wrapper for the Windows MFC as described in *Dealing with binary only dlls*. But in practice it does not seem to be a realistic approach for the MFC:

- the huge number of APIs makes writing the wrapper a big task in itself.
- furthermore the MFC contain a huge number of APIs which are tricky to deal with when making a wrapper.
- even once you have written the wrapper you will need to modify the MFC headers so that the compiler does not choke on them.
- a big part of the MFC code is actually in your application in the form of macros. This means even more of the MFC headers have to actually work to in order for you to be able to compile an MFC based application.

This is why this guide includes a section dedicated to helping you compile the MFC with Winelib.

Legal issues

(Extracted from the HOWTO-Winelib written by Wilbur Dale <wilbur.dale@lumin.nl>)

The purpose of this section is to make you aware of potential legal problems. Be sure to read your licenses and to consult your lawyers. In any case you should not consider the remainder of this section to be authoritative since it has not been written by a lawyer.

Well, let's try to have a look at the situation anyway.

During the compilation of your program, you will be combining code from several sources: your code, Winelib code, Microsoft MFC code, and possibly code from other vendor sources. As a result, you must ensure that the licenses of all code sources are obeyed. What you are allowed and not allowed to do can vary depending on how you compile your program and if you will be distributing it. For example, if you are releasing your code under the GPL, you cannot link your code to MFC code because the GPL requires that you provide ALL sources to your users. The MFC license forbids you from distributing the MFC source so you cannot both distribute your program and comply with the GPL license. On the other hand, if your code is released under the LGPL, you cannot statically link your program to the MFC and distribute it, but you can dynamically link your LGPL code and the MFC library and distribute it.

Wine/Winelib is distributed under an X11-like license. It places few restrictions on the use and distribution of Wine/Winelib code. I doubt the Wine license will cause you any problems. On the other hand, MFC is distributed under a very restrictive license and the restrictions vary from version to version and between service packs. There are basically three aspects you must be aware of when using the MFC.

First you must legally get MFC source code on your computer. The MFC source code comes as a part of Visual Studio. The license for Visual Studio implies it is a single product that can not be broken up into its components. So the cleanest way to get MFC on your system is to buy Visual Studio and install it on a dual boot Linux box.

Then you must check that you are allowed to recompile MFC on a non-Microsoft operating system! This varies with the version of MFC. The MFC license from Visual Studio 6.0 reads in part:

> 1.1 General License Grant. Microsoft grants to you as an individual, a personal, nonexclusive license to make and use copies of the SOFTWARE PRODUCT for the sole purposes of designing, developing, and testing your software product(s) that are designed to operate in conjunction with any Microsoft operating system product. [Other unrelated stuff deleted.]

So it appears you cannot even compile MFC for Winelib using this license. Fortunately the Visual Studio 6.0 service pack 3 license reads (the Visual Studio 5.0 license is similar):

> 1.1 General License Grant. Microsoft grants to you as an individual, a personal, nonexclusive license to make and use copies of the SOFTWARE PRODUCT for the purpose of designing, developing, and testing your software product(s). [Other unrelated stuff deleted]

So under this license it appears you can compile MFC for Winelib.

Finally you must check whether you have the right to distribute an MFC library. Check the relevant section of the license on “redistributables and your redistribution rights”. The license seems to specify that you only have the right to distribute binaries of the MFC library if it has no debug information and if you distribute it with an application that provides significant added functionality to the MFC library.

Compiling the MFC

Here is a set of recommendations for getting the MFC compiled with WineLib:

We recommend running winemaker in ’`--interactive`’ mode to specify the right options for the MFC and the ATL part (to get the include paths right, to not consider the MFC MFC-based, and to get it to build libraries, not executables).

Then when compiling it you will indeed need a number of `_AFX_NO_XXX` macros. But this is not enough and there are other things you will need to ’`#ifdef`-out’. For instance Wine’s richedit support is not very good. Here are the AFX options I use:

```
#define _AFX_PORTABLE
#define _FORCENAMELESSUNION
#define _AFX_NO_DAO_SUPPORT
#define _AFX_NO_DHTML_SUPPORT
#define _AFX_NO_OLEDB_SUPPORT
#define _AFX_NO_RICHEDIT_SUPPORT
```

You will also need custom ones for `CMonikerFile`, `OleDB`, `HtmlView`, ...

We recommend using Wine’s msvcrt headers (`-isystem $(WINE_INCLUDE_ROOT)/msvcrt`), though it means you will have to temporarily disable winsock support (`#ifdef` it out in `windows.h`).

You should use g++ compiler more recent than g++ 2.95. g++ 2.95 does not support unnamed structs while the more recent ones do, and this helps a lot. Here are the options worth mentioning:

- `-fms-extensions` (helps get more code to compile)
- `-fshort-wchar -DWINE_UNICODE_NATIVE` (helps with Unicode support)
- `-DICOM_USE_COM_INTERFACE_ATTRIBUTE` (to get the COM code to work)

When you first reach the link stage you will get a lot of undefined symbol errors. To fix these you will need to go back to the source and `#ifdef`-out more code until you reach a ’closure’. There are also some files that don’t need to be compiled.

Maybe we will have ready-made makefile here someday...

Using the MFC

Specific winemaker options, the configure options, the initialization problem...

Chapter 5. Dealing with binary only dlls

Introduction

For one reason or another you may find yourself with a Linux shared library that you want to use as if it was a Windows Dll. There are various reasons for this including the following:

- You are porting a large application that uses several third-party libraries. One is available on Linux but you are not yet ready to link to it directly as a Linux shared library.
- (The ODBC interface in WINE). There is a well-defined interface available and there are several Linux solutions that are available for it.

The process for dealing with these situations is actually quite simple. You need to write a spec file that will describe the library's interface in the same format as a Dll (primarily what functions it exports). Also you will want to write a small wrapper around the library. We combine these to form a Wine builtin Dll that links to the Linux library.

In this section we will look at two examples. The first example is extremely simple and leads into the subject in "baby steps". The second example is the ODBC interface proxy in Wine. The files to which we will refer for the ODBC example are currently in the `dlls/odbc32` directory of the Wine source.

The first example is based very closely on a real case (the names of the functions etc. have been changed to protect the innocent). A large Windows application includes a DLL that links to a third-party DLL. For various reasons the third-party DLL does not work too well under Wine. However the third-party DLL is also available for the Linux environment. Conveniently the DLL and Linux shared library export only a small number of functions and the application only uses one of those.

Specifically, the application calls a function:

```
signed short WINAPI MyWinFunc (unsigned short a, void *b, void *c,
        unsigned long *d, void *e, unsigned char f, char g,
        unsigned char *h);
```

and the linux library exports a corresponding function:

```
signed short MyLinuxFunc (unsigned short a, void *b, void *c,
        unsigned short *d, void *e, char g, unsigned char *h);
```

Writing the spec file

Start by writing the spec file. This file will describe the interface as if it was a dll. See elsewhere for the details of the format of a spec file.

In the simple example we want a Wine builtin Dll that corresponds to the MyWin Dll. The spec file is `libMyWin.spec` and looks like this.

```
#
# File: libMyWin.spec
#
# some sort of copyright
#
# Wine spec file for the libMyWin builtin library (a minimal wrapper around the
# linux library libMyLinux)
#
```

```
# For further details of wine spec files see the Winelib documentation at
# www.winehq.com

name    MyWin
type    win32
mode    dll

2 stdcall _MyWinFunc@32 (long ptr ptr ptr ptr long long ptr) MyProxyWinFunc

# End of file
```

Notice that the arguments are flagged as long even though they are smaller than that.

In the case of the ODBC example you can see this in the file `odbc32.spec`.

How to deal with C++ APIs

names are mangled, how to demangle them, how to call them

Writing the wrapper

Firstly we will look at the simple example. The main complication of this case is the slightly different argument lists. The f parameter does not have to be passed to the Linux function and the d parameter (theoretically) has to be converted between unsigned long * and unsigned short *. Doing this ensures that the "high" bits of the returned value are set correctly.

```
/*
 * File: MyWin.c
 *
 * Copyright (c) The copyright holder.
 *
 * Basic WINE wrapper for the linux <3rd party library> so that it can be
 * used by <the application>
 *
 * Currently this file makes no attempt to be a full wrapper for the <3rd
 * party library>; it only exports enough for our own use.
 *
 * Note that this is a Unix file; please don't go converting it to DOS format
 * (e.g. converting line feeds to Carriage return/Line feed).
 *
 * This file should be built in a Wine environment as a WineLib library,
 * linked to the Linux <3rd party> libraries (currently libxxxx.so and
 * libyyyy.so)
 */

#include < <3rd party linux header> >
#include <windef.h> /* Part of the Wine header files */

signed short WINAPI MyProxyWinFunc (unsigned short a, void *b, void *c,
                unsigned long *d, void *e, unsigned char f, char g,
                unsigned char *h)
/* This declaration is as defined in the spec file.  It is deliberately not
 * specified in terms of <3rd party> types since we are messing about here
 * between two operating systems (making it look like a Windows thing when
 * actually it is a Linux thing).  In this way the compiler will point out any
 * inconsistencies.
```

```
 * For example the fourth argument needs care
 */
{
    unsigned short d1;
    signed short ret;

    d1 = (unsigned short) *d;
    ret = <3rd party linux function> (a, b, c, &d1, e, g, h);
    *d = d1;

    return ret;
}

/* End of file */
```

For a more extensive case we can use the ODBC example. This is implemented as a header file (`proxyodbc.h`) and the actual C source file (`proxyodbc.c`). Although the file is quite long it is extremely simple in structure.

The MAIN_OdbcInit function is the function that was named in the spec file as the init function. On the process attach event the function dynamically links to the desired Linux ODBC library (since there are several available) and builds a list of function pointers. It unlinks on the process detach event.

Then each of the functions simply calls the appropriate Linux function through the function pointer that was set up during initialisation.

Building

So how dow we actually build the Wine builtin Dll? The easiest way is to get Winemaker to do the hard work for us. For the simple example we have two source files (the wrapper and the spec file). We also have the 3rd party header and library files of course.

Put the two source files in a suitable directory and then use winemaker to create the build framework, including configure script, makefile etc. You will want to use the following options of winemaker:

- --nosource-fix and --nogenerate-specs (requires winemaker version 0.5.8 or later) to ensure that the two files are not modified. (If using an older version of winemaker then make the two files readonly and ignore the complaints about being unable to modify them).
- --dll --single-target MyWin --nomfc to specify the target
- -DMightNeedSomething -I3rd_party_include -L3rd_party_lib -lxxxx -lyyyy where these are the locations of the header files etc.

After running winemaker I like to edit the Makefile.in to add the line CEXTRA = -Wall just before the DEFINES =.

Then simply run the configure and make as normal (described elsewhere).

Chapter 6. Packaging your Winelib application

Selecting which libraries to deliver, how to avoid interference with other Winelib applications, how to play nice with other Winelib applications

Wine Packagers Guide

Wine Packagers Guide

Table of Contents

Chapter 1. Preface

Authors

Written by Marcus Meissner <Marcus.Meissner@caldera.de> Updated by Jeremy White <jwhite@codeweavers.com> Updated by Andreas Mohr <amohr@codeweavers.com>

Document Revision Date

The information contained in this document is extremely time sensitive. *It is vital that a packager stay current with changes in Wine.* Changes to this document could be tracked e.g. by viewing its CVS log. Due to Wine's fast development, a recent revision date does not necessarily indicate that this document is 100% on par with what Wine's full installation requirements are (especially whenever lazy developers don't properly update the documentation to include info about new features they implemented).

This document was last revised on November 14, 2001.

Terms used in this document

There are several terms and paths used in this document as place holders for configurable values. Those terms are described here.

1. WINECONFDIR

   WINECONFDIR is the user's Wine configuration directory. This is almost always ~/.wine, but can be overridden by the user by setting the WINECONFDIR environment variable.

2. PREFIX

   PREFIX is the prefix used when selecting an installation target. The current default is /usr. This results in binary installation into /usr/bin, library installation into /usr/wine/lib, and so forth. This value can be overridden by the packager. In fact, FHS 2.1 (http://www.pathname.com/fhs/) specifications suggest that a better prefix is /opt/wine. Ideally, a packager would also allow the installer to override this value.

3. ETCDIR

   ETCDIR is the prefix that Wine uses to find the global configuration directory. This can be changed by the configure option sysconfdir. The current default is /etc.

4. WINDOWSDIR

   WINDOWSDIR is an important concept to Wine. This directory specifies what directory corresponds to the root Windows directory (e.g. C:\WINDOWS).

   This directory is specified by the user, in the user's configuration file.

   Generally speaking, this directory is either set to point at an empty directory, or it is set to point at a Windows partition that has been mounted through the vfat driver.

   It is extremely important that the packager understand the importance of WINDOWSDIR and convey this information and choice to the end user.

Chapter 2. Introduction

This document attempts to establish guidelines for people making binary packages of Wine.

It expresses the basic principles that the Wine developers have agreed should be used when building Wine. It also attempts to highlight the areas where there are different approaches to packaging Wine, so that the packager can understand the different alternatives that have been considered and their rationales.

Goals

An installation from a Wine package should:

- Install quickly and simply.

  The initial installation should require no user input. An rpm -i wine.rpm or apt-get install wine should suffice for initial installation.

- Work quickly and simply

  The user should be able to launch Solitaire within minutes of downloading the Wine package.

- Comply with Filesystem Hierarchy Standard

  A Wine installation should, as much as possible, comply with the FHS standard (http://www.pathname.com/fhs/).

- Preserve flexibility

  None of the flexibility built into Wine should be hidden from the end user.

- 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.
- Reduce support requirements.

  A packaged version of Wine should be sufficiently easy to use and have quick and easy access to FAQs and documentation such that requests to the newsgroup and development group go down. Further, it should be easy for users to capture good bug reports.

Requirements

Successfully installing Wine requires:

- Much thought and work from the packager (1x)
- A configuration file

  Wine will not run without a configuration file. Further, no default is currently provided by Wine. Some packagers may attempt to provide (or dynamically generate) a default configuration file. Some packagers may wish to rely on winesetup to generate the configuration file.

- 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`.
- An initial set of registry entries.

  The current Wine standard is to use the regapi tool against the 'winedefault.reg' file to generate a default registry.

  There are several other choices that could be made; registries can be imported from a Windows partition. At this time, Wine does not completely support a complex multi-user installation ala Windows NT, but it could fairly readily.

- Some special `.dll` and `.exe` files in the `windows\system` directory, since applications directly check for their presence.

Chapter 3. Wine Components

This section lists all files that pertain to Wine.

Wine Static and Shareable Files

At the time of this writing, almost all of the following components are installed through a standard 'make install' of Wine. Exceptions from the rule are noted.

> **Caution**
>
> It is vital that a packager check for changes in Wine. This list will likely be out of date by the time this document is committed to CVS.

1.

Executable Files

`wine`

The main Wine executable. This program will load a Windows binary and run it, relying upon the Wine shared object libraries.

`wineserver`

The Wine server is critical to Wine; it is the process that coordinates all shared Windows resources.

`winebootup`

Winelib app to be found in programs/. It'll be called by the winelauncher wine wrapper startup script for every first-time wine invocation. Its purpose is to process all Windows startup autorun mechanisms, such as wininit.ini, win.ini Load=/Run=, registry keys: RenameFiles/Run/RunOnce*/RunServices*, Startup folders.

`wineclipsrv`

The Wine Clipboard Server is a standalone XLib application whose purpose is to manage the X selection when Wine exits.

`winedbg`

Winedbg is the Wine built in debugger.

`winelauncher`

(not getting installed via make install) A wine wrapper shell script that intelligently handles wine invocation by informing the user about what's going on, among other things. To be found in tools/ directory. Use of this wrapper script instead of directly using wine is strongly encouraged, as it not only improves the user interface, but also adds important functionality to wine, such as session bootup/startup actions. If you intend to use this script, then you might want to rename the wine executable to e.g. wine.bin and winelauncher to wine. the *WINECONFDIR*/config file.

`winesetup`

This is a Tcl/Tk based front end that provides a user friendly tool to edit and configure the *WINECONFDIR*/config file.

`wineshelllink`

This shell script can be called by Wine in order to propagate Desktop icon and menu creation requests out to a GNOME or KDE (or other Window Managers).

`winebuild`

Winebuild is a tool used for Winelib applications (and by Wine itself) to allow a developer to compile a .spec file into a .spec.c file.

`wmc`

The wmc tools is the Wine Message Compiler. It allows Windows message files to be compiled into a format usable by Wine.

`wrc`

The wrc tool is the Wine Resource Compiler. It allows Winelib programmers (and Wine itself) to compile Windows style resource files into a form usable by Wine.

`fnt2bdf`

The fnt2bdf utility extracts fonts from .fnt or .dll files and stores them in .bdf format files.

`dosmod`

DOS Virtual Machine.

`uninstaller`

(not getting installed via make install) A Winelib program to uninstall installed Windows programs. To be found in the programs/ source directory. This program can be used to uninstall most Windows programs (just like the Add/Remove Programs item in Windows) by taking the registry uninstall strings that get created by installers such as InstallShield or WISE. In binary packages, it should probably be renamed to something like wine-uninstaller for consistency's sake.

2. Shared Object Library Files

This list is NOT necessarily current !

advapi32.dll.so	imm32.dll.so	msdmo.dll.so	qcap.dll.so	ver.dll.so
avicap32.dll.so	joystick.drv.so	msg711.drv.so	quartz.dll.so	version.dll.so
avifil32.dll.so	kernel32.dll.so	msimg32.dll.so	rasapi16.dll.so	w32skrnl.dll.so
avifile.dll.so	keyboard.dll.so	msnet32.dll.so	rasapi32.dll.so	w32sys.dll.so
comctl32.dll.so	krnl386.exe.so	msrle32.dll.so	riched32.dll.so	win32s16.dll.so
comdlg32.dll.so	libgdi32.dll.so	msvcrt.dll.so	rpcrt4.dll.so	win87em.dll.so
comm.dll.so	libkernel32.dll.so	msvcrt20.dll.so	serialui.dll.so	winaspi.dll.so
commdlg.dll.so	libntdll.dll.so	msvfw32.dll.so	setupapi.dll.so	windebug.dll.so
compobj.dll.so	libuser32.dll.so	msvideo.dll.so	setupx.dll.so	winearts.drv.so
crtdll.dll.so	libwine.so	netapi32.dll.so	shdocvw.dll.so	winedos.dll.so
crypt32.dll.so	libwine_tsx11.so	ntdll.dll.so	shell.dll.so	wineoss.drv.so
dciman32.dll.so	libwine_unicode.so	odbc32.dll.so	shell32.dll.so	wineps.dll.so
ddeml.dll.so	libwinspool.drv.so	ole2.dll.so	shfolder.dll.so	wineps16.dll.so
ddraw.dll.so	lz32.dll.so	ole2conv.dll.so	shlwapi.dll.so	wing.dll.so
devenum.dll.so	lzexpand.dll.so	ole2disp.dll.so	sound.dll.so	wininet.dll.so
dinput.dll.so	mapi32.dll.so	ole2nls.dll.so	sti.dll.so	winmm.dll.so
dispdib.dll.so	mcianim.drv.so	ole2prox.dll.so	storage.dll.so	winnls.dll.so
display.dll.so	mciavi.drv.so	ole2thk.dll.so	stress.dll.so	winnls32.dll.so
dplay.dll.so	mcicda.drv.so	ole32.dll.so	system.dll.so	winsock.dll.so
dplayx.dll.so	mciseq.drv.so	oleaut32.dll.so	tapi32.dll.so	winspool.drv.so
dsound.dll.so	mciwave.drv.so	olecli.dll.so	toolhelp.dll.so	wintrust.dll.so
gdi.exe.so	midimap.drv.so	olecli32.dll.so	ttydrv.dll.so	wnaspi32.dll.so

gdi32.dll.so	mmsystem.dll.so	oledlg.dll.so	twain_32.dll.so	wow32.dll.so
glu32.dll.so	mouse.dll.so	olepro32.dll.so	typelib.dll.so	wprocs.dll.so
icmp.dll.so	mpr.dll.so	olesvr.dll.so	url.dll.so	ws2_32.dll.so
imaadp32.acm.so	msacm.dll.so	olesvr32.dll.so	urlmon.dll.so	wsock32.dll.so
imagehlp.dll.so	msacm.drv.so	opengl32.dll.so	user.exe.so	x11drv.dll.so
imm.dll.so	msacm32.dll.so	psapi.dll.so	user32.dll.so	

3. Man Pages
   wine.man
   wine.conf.man
   wmc.man
   wrc.man

4. Include Files

   This list is NOT necessarily current !

basetsd.h	lzexpand.h	rpc.h	wine/obj_channel.h	wine/obj_shellfolder.h
cderr.h	mapidefs.h	servprov.h	wine/obj_clientserver.h	wine/obj_shelllink.h
cguid.h	mcx.h	shellapi.h	wine/obj_commdlgbrowser.h	wine/obj_shellview.h
commctrl.h	mmreg.h	shlguid.h	wine/obj_connection.h	wine/obj_storage.h
commdlg.h	mmsystem.h	shlobj.h	wine/obj_contextmenu.h	wine/unicode.h
compobj.h	msacm.h	shlwapi.h	wine/obj_control.h	winerror.h
d3d.h	ntsecapi.h	sql.h	wine/obj_dataobject.h	wingdi.h
d3dcaps.h	oaidl.h	sqlext.h	wine/obj_dockingwindowframe.h	wininet.h
d3dtypes.h	objbase.h	sqltypes.h	wine/obj_dragdrop.h	winioctl.h
d3dvec.inl	objidl.h	storage.h	wine/obj_enumidlist.h	winnetwk.h
dde.h	ocidl.h	tapi.h	wine/obj_errorinfo.h	winnls.h
ddeml.h	ole2.h	tlhelp32.h	wine/obj_extracticon.h	winnt.h
ddraw.h	ole2ver.h	unknwn.h	wine/obj_inplace.h	winreg.h
digitalv.h	oleauto.h	urlmon.h	wine/obj_marshal.h	winresrc.h
dinput.h	olectl.h	ver.h	wine/obj_misc.h	winsock.h
dispdib.h	oledlg.h	vfw.h	wine/obj_moniker.h	winsock2.h
dlgs.h	oleidl.h	winbase.h	wine/obj_oleaut.h	winspool.h
docobj.h	poppack.h	wincon.h	wine/obj_olefont.h	winsvc.h
dplay.h	prsht.h	wincrypt.h	wine/obj_oleobj.h	winuser.h
dplobby.h	psapi.h	windef.h	wine/obj_oleundo.h	winver.h
dsound.h	pshpack1.h	windows.h	wine/obj_oleview.h	wnaspi32.h
guiddef.h	pshpack2.h	windowsx.h	wine/obj_picture.h	wownt32.h
imagehlp.h	pshpack4.h	wine/exception.h	wine/obj_property.h	wtypes.h
imm.h	pshpack8.h	wine/icmpapi.h	wine/obj_propertystorage.h	zmouse.h
initguid.h	ras.h	wine/ipexport.h	wine/obj_queryassociations.h	
instance.h	regstr.h	wine/obj_base.h	wine/obj_shellbrowser.h	
lmcons.h	richedit.h	wine/obj_cache.h	wine/obj_shellextinit.h	

5. Documentation files.

   At the time of this writing, I do not have a definitive list of documentation files to be installed. However, they do include the HTML files generated from the SGML in the Wine CVS tree.

Dynamic Wine Files

Wine also generates and depends on a number of dynamic files, including user configuration files and registry files.

At the time of this writing, there was not a clear consensus of where these files should be located, and how they should be handled. This section attempts to explain the alternatives clearly.

1.

Configuration File

WINECONFDIR/config

This file is the user local Wine configuration file. At the time of this writing, if this file exists, then no other configuration file is loaded.

ETCDIR/wine.conf

This is the global Wine configuration file. It is only used if the user running Wine has no local configuration file.

Some packagers feel that this file should not be supplied, and that only a wine.conf.default should be given here.

Other packagers feel that this file should be the predominant file used, and that users should only shift to a local configuration file if they need to. An argument has been made that the local configuration file should inherit the global configuration file. At this time, Wine does not do this; please refer to the WineHQ discussion archives for the debate concerning this.

This debate is addressed more completely below, in *Packaging Strategies*.

2. Registry Files

In order to replicate the Windows registry system, Wine stores registry entries in a series of files. For an excellent overview of this issue, read this Wine Weekly News feature. (http://www.winehq.com/News/2000-25.html#FTR)

The bottom line is that, at Wine server startup, Wine loads all registry entries into memory to create an in memory image of the registry. The order of files which Wine uses to load registry entries is extremely important, as it affects what registry entries are actually present. The order is roughly that .dat files from a Windows partion are loaded, then global registry settings from *ETCDIR*, and then finally local registry settings are loaded from *WINECONFDIR* . As each set are loaded, they can override the prior entries. Thus, the local registry files take precedence.

Then, at exit (or at periodic intervals), Wine will write either all registry entries (or, with the default setting) changed registry entries to files in the *WINECONFDIR*.

WINECONFDIR/system.reg

This file contains the user's local copy of the HKEY_LOCAL_MACHINE registry hive. In general use, it will contain only changes made to the default registry values.

WINECONFDIR/user.reg

This file contains the user's local copy of the HKEY_CURRENT_USER registry hive. In general use, it will contain only changes made to the default registry values.

WINECONFDIR/userdef.reg

This file contains the user's local copy of the HKEY_USERS\.Default registry hive. In general use, it will contain only changes made to the default registry values.

WINECONFDIR/wine.userreg

This file is being deprecated. It is only read if there is no user.reg or wine.userreg, and it supplied the contents of HKEY_USERS.

ETCDIR/wine.systemreg

This file contains the global values for HKEY_LOCAL_MACHINE. The values in this file can be overridden by the user's local settings.

Note: The location of this directory is hardcoded within wine, generally to /etc. This will hopefully be fixed at some point in the future.

ETCDIR/wine.userreg

This file contains the global values for HKEY_USERS. The values in this file can be overridden by the user's local settings. This file is likely to be deprecated in favor of a global wine.userdef.reg that will only contain HKEY_USERS/.Default.

3.

Other files in *WINECONFDIR*

WINECONFDIR/wineserver-[hostname]

This directory contains files used by Wine and the Wineserver to communicate. A packager may want to have a facility for the user to erase files in this directory, as a crash in the wineserver resulting in a bogus lock file can render wine unusable.

WINECONFDIR/cachedmetrics.[display]

This file contains font metrics for the given X display. Generally, this cache is generated once at Wine start time.

Important Files from a Windows Partition

Wine has the ability to use files from an installation of the actual Microsoft Windows operating system. Generally these files are loaded on a VFAT partition that is mounted under Linux.

This is probably the most important configuration detail. The use of Windows registry and DLL files dramatically alters the behaviour of Wine. If nothing else, pacakager have to make this distinction clear to the end user, so that they can intelligently choose their configuration.

1.

Registry Files

[WINDOWSDIR]/system32/system.dat

[WINDOWSDIR]/system32/user.dat

[WINDOWSDIR]/win.ini

2. Windows Dynamic Link Libraries ([WINDOWSDIR]/system32/*.dll)

   Wine has the ability to use the actual Windows DLL files when running an application. An end user can configure Wine so that Wine uses some or all of these DLL files when running a given application.

Chapter 4. Packaging Strategies

There has recently been a lot of discussion on the Wine development mailing list about the best way to build Wine packages.

There was a lot of discussion, and several diverging points of view. This section of the document attempts to present the areas of common agreement, and also to present the different approaches advocated on the mailing list.

Distribution of Wine into packages

The most basic question to ask is given the Wine CVS tree, what physical files are you, the packager, going to produce? Are you going to produce only a wine.rpm (as Marcus has done), or are you going to produce 6 Debian files (libwine, libwine-dev, wine, wine-doc, wine-utils and winesetuptk) as Ove has done?

At this point, there is no consensus amongst the wine-devel community on this subject.

Where to install files

This question is not really contested. It will vary by distribution, and is really up to the packager. As a guideline, the current 'make install' process seems to behave such that if we pick a single *PREFIX*, then :

1. all binary files go into *PREFIX*/bin,
2. all library files go into *PREFIX*/lib,
3. all include files go into *PREFIX*/include,
4. all documentation files go into *PREFIX*/doc/wine,
5. and man pages go into *PREFIX*/man,

Refer to the specific information on the Debian package and the OpenLinux package for specific details on how those packages are built.

You might also want to use the wine wrapper script winelauncher that can be found in tools/ directory, as it has several important advantages over directly invoking the wine binary. See the Executable Files section for details.

The question of /opt/wine

The FHS 2.1 specification suggests that Wine as a package should be installed to /opt/wine. None of the existing packages follow this guideline (today; check again tomorrow).

What files to create

After installing the static and shareable files, the next question the packager needs to ask is how much dynamic configuration will be done, and what configuration files should be created.

There are several approaches to this:

1. Rely completely on user file space - install nothing

   This approach relies upon the new winesetup utility and the new ability of Wine to launch winesetup if no configuration file is found. The basic concept is that no global configuration files are created at install time. Instead, Wine configuration files are created on the fly by the winesetup program when Wine is invoked. Further, winesetup creates default Windows directories and paths that are stored completely in the user's *WINECONFDIR*.

This approach has the benefit of simplicity in that all Wine files are either stored under /opt/wine or under ~/.wine. Further, there is only ever one Wine configuration file.

This approach, however, adds another level of complexity. It does not allow Wine to run Solitaire 'out of the box'; the user must run the configuration program first. Further, winesetup requires Tcl/Tk, a requirement not beloved by some. Additionally, this approach closes the door on multi user configurations and presumes a single user approach.

2. Build a reasonable set of defaults for the global wine.conf, facilitate creation of a user's local Wine configuration.

   This approach, best shown by Marcus, causes the installation process to auto scan the system, and generate a global wine.conf file with best guess defaults. The OpenLinux packages follow this behaviour.

   The keys to this approach are always putting an existing Windows partition into the path, and being able to run Solitaire right out of the box. Another good thing that Marcus does is he detects a first time installation and does some clever things to improve the user's Wine experience.

   A flaw with this approach, however, is it doesn't give the user an obvious way to choose not to use a Windows partition.

3. Build a reasonable set of defaults for the global wine.conf, and ask the user if possible

   This approach, demonstrated by Ove, causes the installation process to auto scan the system, and generate a global wine.conf file with best guess defaults. Because Ove built a Debian package, he was able to further query debconf and get permission to ask the user some questions, allowing the user to decide whether or not to use a Windows partition.

What to put into the wine config file

The next hard question is what the Wine config should look like. The current best practices seems to involve using drives from M to Z.

> **Caution**
>
> This isn't done yet! Fix it, Jer!

Chapter 5. Implementation

OpenLinux Sample

1. 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 dlls/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 *`WINECONFDIR`*`/user.reg` and *`WINECONFDIR`*`/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
```

2. 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 distribution's default floppy mountpoint here.

       `Path=/auto/floppy`

     - C for the `C:\` directory. Here we use the user's home directory, 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 distribution's 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 occasionally complain about in the newsgroup and the irc channel. 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 user's
  home directory)
System=c:\windows\system\ (the windows/system subdirectory in the user's
  home directory)
Path=c:\windows;c:\windows\system;c:\windows\system32;w:\;w:\system;w:\system32;
; 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.

3. 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. Also make sure that the kernel you use includes the Joliet CD-ROM support, for the very same reasons as given above (no long filenames due to missing Joliet, files not found).

4. 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 one 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 user's windows directories and in the global windows directories and replace same files by symlinks (to avoid diskspace problems).

Sample `wine.ini` for OpenLinux 2.x (outdated, for review purposes only !):

```
;;
;; 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 user's 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
Type=hd
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.
;
Windows=c:\windows
;
; 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
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, 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>
```

Chapter 6. Work to be done

In preparing this document, it became clear that there were still a range of action items to be done in Wine that would improve this packaging process. For lack of a better place, I record them here. *This list is almost certain to be obsolete; check bugzilla for a better list.*

1. Remove duplication of code between winesetup and wineconf/wineinstall.

   Currently, winesetup duplicates all of the code contained in wineconf.

   Instead, wineconf should be improved to generate the new style config file, and then winesetup should rely on wineconf to generate the default configuration file.

   Similarly, there is functionality such as creating the default registry files that is now done by both winesetup and wineinstall.

   At this time, it seems like the right thing to do is to break up or parameterize wineinstall, so that it can be used for single function actions, and then have winesetup call those functions.

2. Enhance winesetup to support W: drive generation.

   The best practices convention now seems to be to generate a set of drives from M: through W:. At this point, winesetup does not generate a default wine config file that follows these conventions. It should.

3. Enhance Wine to allow more dynamic switching between the use of a real Windows partition and an empty one.
4. Write a winelauncher utility application.

   Currently, Wine really requires a user to launch it from a command line, so that the user can look for error messages and warnings. However, eventually, we will want users to be able to launch Wine from a more friendly GUI launcher. The launcher should have the ability to allow the end user to turn on debugging messages and capture those traces for bug reporting purposes. Also, if we make it possible to switch between use of a Windows partition or not automatically, that option should be controlled here.

5. Get Marcus’s winesetup facilities into CVS

   Along the lines of the changes to winesetup, and the consolidation of wineconf and wineinstall, we should extract the good stuff from Marcus’s winesetup script, and get it into CVS. Again, perhaps we should have a set of scripts that perform discrete functions, or maybe one script with parameters.

6. Finish this document

   This document is pretty rough itself. Many hard things aren’t addressed, and lots of stuff was missed.