

Wine User Guide

Wine User Guide

Table of Contents

1. Introduction	7
What is Wine?.....	7
Wine Requirements and Features	8
2. Getting Wine	11
The Many Forms of Wine.....	11
Getting Wine for a Debian System	11
Getting Wine for a Red Hat System	11
Getting Wine for Other Distributions	12
3. Installing/compiling Wine.....	13
WWN #52 Feature: Replacing Windows	13
Installing Wine Without Windows.....	14
Installing Wine Using An Existing Windows Partition As Base.....	15
Dealing With FAT/VFAT Partitions.....	16
SCSI Support	18
4. Configuring Wine	21
General Configuration	21
Configuring the x11drv Driver	32
The Registry.....	35
Setting the windows and DOS version value that's passed to programs	37
Drive labels and serial numbers with wine.....	38
DLL configuration	40
Dealing with Fonts.....	44
Printing in Wine.....	48
Win95/98 Look.....	51
Keyboard.....	51
Using ODBC.....	54
5. Running Wine	55
How to run Wine.....	55
Command-Line Options	55
Setting Windows/DOS environment variables	58
6. Troubleshooting / Reporting bugs.....	59
What to do if some program still doesn't work ?	59
How To Report A Bug	60

Chapter 1. Introduction

What is Wine?

Written by John R. Sheets <jsheets@codeweavers.com>
Modified by Dustin Navea (<mailto:Speeddymon@yahoo.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 (<http://www.tldp.org/FAQ/Linux-FAQ/index.html>), 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, yet.¹ However, Wine makes it possible to run native Windows applications alongside native Linux applications on any Unix-like 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) (<http://www.xfree86.org/#whatis>) 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.

Burning questions and comments

If during reading this document there is something you can't figure out, or think could be explained better, or that should have been included, please immediately mail to either the WineHQ Web-Admin <web-admin@winehq.com> or the wine-devel Mailing List <wine-devel@winehq.com>, or post a bug report to Wine's Bugzilla (<http://bugs.winehq.com/>) to let us know how this document can be improved. Remember, Open Source is "free as in free speech, not as in free beer": it can only work in the case of very active involvement of its users !

Wine Requirements and Features

Written by Andreas Mohr <amohr@codeweavers.com>
Modified by Dustin Navea (mailto:Speeddymon@yahoo.com)

System requirements

In order to run Wine, you need the following:

- - A computer ;-)
 - Wine: only PCs \geq i386 are supported at the moment.
 - WineLib: selected other platforms are supported, but can be tricky.
- A UNIX-like operating system such as Linux, *BSD, Solaris x86, ReactOS, Cygwin
- \geq 32MB of RAM. Everything below is pretty much unusable. \geq 96 MB is needed for "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 (ttydrv) 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/XP 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

- 32 bit graphical coordinates for CAD applications, 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 should be in luck: A prepackaged version of Wine should already exist for your system. The following 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, Red Hat, and other distributions.

If you're not lucky enough to have a package available for your operating system, or if you'd prefer a newer version of Wine than already exists as a package, you will 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 skills. Working from the source is covered in the Wine Developer's Guide.

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 -  
Password:  
# cd /home/user  
# dpkg -i wine_0.0.20021031-1.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 Red Hat System

Red Hat/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.

Of course now that you have the RPM package, you may be wondering "What in the world do I do with this thing?".

The easiest way to install an RPM is to make sure that you have not previously installed wine (perhaps, when you installed linux) and then switch to the directory you downloaded the rpm file to. Once there, type this one command as root:

```
# rpm -ivh wine-20020605-2.i386.rpm
```

You may also want to install the wine-devel package.

Getting Wine for Other Distributions

The first place you should look if your system isn't Debian or Red Hat 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.

NOTE: If you are running a Mandrake system, please see the page on how to get wine for a Redhat system, as Mandrake is based on Redhat.

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 [here](#).

Directory Structure

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

<code>C:\</code>	Root directory of primary disk drive
<code>Windows\</code>	Windows directory, containing <code>.INI</code> files, accessories, etc.
<code>System\</code>	Win3.x/95/98/ME directory for common DLLs WinNT/2000 directory for common 16-bit DLLs
<code>System32\</code>	WinNT/2000 directory for common 32-bit DLLs
<code>Start Menu\</code>	Program launcher directory structure
<code>Programs\</code>	Program launcher links (<code>.LNK</code> 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 configured `c:\windows\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 one of the Windows DLL archive sites that can be found via internet search engine. Please make sure to obey any licenses on the DLLs you fetch... (some are redistributable, some aren't).

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. Point `[Drive C]` in `~/.wine/config` to the directory where you want `C:` to be. Refer to the `wine.conf` man page for more information. The directory to be used for emulating a `C:` drive will be the base directory for some Windows specific directories created below. Remember to use `"Filesystem" = "win95"!`

2. Within the directory to be used for C:, create empty `windows`, `windows/system`, `windows/Start Menu`, and `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). In one line: `mkdir -p windows windows/system windows/Start\ Menu windows/Start\ Menu/Programs`
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 +loaddll` 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`, `USER/USER32` or `NTDLL`. 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.

Installing Wine Using An Existing Windows Partition As Base

Some people intend to use the data of an existing Windows partition with Wine in order to gain some better compatibility or to run already installed programs in a setup as original as possible. Note that many Windows programs assume that they have full write access to all windows directories. This means that you either have to configure the Windows partition mount point for write permission by your Wine user (see Dealing with FAT/VFAT partitions on how to do that), or you'll have to copy over (some parts of) the Windows partition content to a directory

of a Unix partition and make sure this directory structure is writable by your user. We **HIGHLY DISCOURAGE** people from directly using a Windows partition with write access as a base for Wine !! (some programs, notably Explorer, corrupt large parts of the Windows partition in case of an incorrect setup; you've been warned). Not to mention that NTFS write support in Linux is still very experimental and **DANGEROUS** (in case you're using an NT-based Windows version using the NTFS file system). Thus we advise you to go the Unix directory way.

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/sg1"
```

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	Overrides 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
[Debug]	no	What to do with certain debug messages

Section Name	Needed?	What it Does
[Registry]	no	Specifies locations of windows registry files
[tweak.layout]	recmd	Appearance of wine
[programs]	no	Programs to be run automatically
[Console]	no	Console settings
[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

These sections are supposed to make certain Unix directory locations accessible to Wine as a DOS/Windows drive (drive 'X:') and thus accessible to Windows programs under the drive name you specified. Every DOS/Windows program sort of expects at least a C: drive (and sometimes also an A: floppy drive), so your config file should at least contain the corresponding sections, [Drive C] and [Drive A]. You need to decide on whether you want to use an existing Windows partition as the C drive or whether you want to create your own Wine drive C directory tree somewhere (take care about permissions !). Each drive section may specify up to 6 different settings as explained below.

```
[Drive X]
```

The above line begins the section for a drive whose letter is X (DOS notation: drive 'X:'). You could e.g. create an equivalent to a drive 'C:' under DOS/Windows by using a [Drive C] section name.

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

This specifies the directory where the drive will begin. When Wine is browsing in drive X, it will be able to see the files that are in the directory `/dir/to/path` and below. (note that symlinks to directories won't get included ! see "ShowDirSymlinks" config setting) You can also make use of environment variables like \$HOME here, an example for using a mywinedrive directory in your home dir would be "Path" = "\${HOME}/mywinedrive" Don't forget to leave off the trailing slash!

```
"Type" = "hd|cdrom|network|floppy"
```

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>'".) Usually, you choose "hd" for a drive ("hd" is default anyway).

```
"Label" = "blah"
```

Defines the drive label. Generally only needed for programs that look for a special CD-ROM. The label may be up to 11 characters. Note that the preferred way of managing labels and serial numbers of CD-ROMs and floppies is to give Wine raw device access for reading these on a per-CD case (see "Device" below) instead of hardcoding one specific "Label".

```
"Serial" = "deadbeef"
```

Tells Wine the serial number of the drive. A few programs with intense protection for pirating might need this, but otherwise it's not needed. Up to 8 characters and hexadecimal. Using a "Device" entry instead of hardcoding the "Serial" probably is a smarter choice.

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

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

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.

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.)

```
"Device" = "/dev/xx"
```

Needed for raw device access and label and serial number reading. Use this **ONLY** for floppy and cdrom devices. Using it on Extended2 or other Unix file systems can have dire results (when a windows app tries to do a lowlevel write, they do it in a FAT way – FAT format is completely different from any Unix file system). Also, make sure that you have proper permissions to this device file.

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

Here are a few sample entries:

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

```
[Drive C]
"Path" = "/dos/c"
"Type" = "hd"
"Label" = "Hard Drive"
"Filesystem" = "win95"
```

This is a setup for Drive E, a generic CD-ROM drive:

```
[Drive E]
"Path" = "/mnt/cdrom"
"Type" = "cdrom"
"Label" = "Total Annihilation"
"Filesystem" = "win95"
"Device" = "/dev/cdrom"
```

And here is a setup for Drive A, a generic floppy drive:

```
[Drive A]
"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.

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

This tells Wine and Windows programs where the windows directory is. It is recommended to have this directory somewhere on your configured C drive, and it's also recommended to just call the directory "windows" (this is the default setup on Windows, and some stupid applications might rely on this). So in case you chose a "Windows" setting of "c:\\windows" and you chose to set up a drive C e.g. at /usr/local/wine_c, the corresponding directory would be /usr/local/wine_c/windows. Make one if you don't already have one. NO TRAILING SLASH (NOT C:\\windows\\)! Write access strongly recommended!

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

This sets up where the windows system files are. The Windows system directory should reside below the directory used for the windows setting. Thus when using the example above, the system directory would be /usr/local/wine_c/windows/system. Again, no trailing slash, and write access!

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

This should be the directory you want your temp files stored in, /usr/local/wine_c/temp in our example. Again, no trailing slash, and WRITE ACCESS!!

```
"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. Note that ttydrv is still very lacking, so if it doesn't work, resort to using "xvfb", a virtual X11 server.

```
"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 to work 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.

builtin

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

so

Native ELF libraries. Will not work yet.

elfdll

ELF encapsulated windows DLL's. No longer used, ignored.

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/ .config` or `wine.conf`, 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]
```

```

"rpcrt4"      = "builtin, native"
"oleaut32"    = "builtin, native"
"ole32"       = "builtin, native"
"commdlg"     = "builtin, native"
"comdlg32"    = "builtin, native"
"ver"         = "builtin, native"
"version"     = "builtin, native"
"shell"       = "builtin, native"
"shell32"     = "builtin, native"
"shfolder"    = "builtin, native"
"shlwapi"     = "builtin, native"
"shdocvw"     = "builtin, native"
"lzexpand"   = "builtin, native"
"lz32"        = "builtin, native"
"comctl32"    = "builtin, native"
"commctrl"    = "builtin, native"
"advapi32"    = "builtin, native"
"crtdll"      = "builtin, native"
"mpr"         = "builtin, native"
"winspool.drv" = "builtin, native"
"ddraw"       = "builtin, native"
"dinput"      = "builtin, native"
"dsound"      = "builtin, native"
"opengl32"    = "builtin, native"
"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"
; you can specify applications too
"notepad.exe" = "native, builtin"
; default for all other dlls
"*" = "native, builtin"

```

Note: If loading of the libraries that are listed first fails, wine will just go on by using 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 (<code>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".

```
"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 the Fonts chapter.

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/ttySY"
```

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/ttyS0"
```

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

The `[paralleports]` 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 [Debug], [Registry], [tweak.layout], and [programs] Sections

[Debug] 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. (In extreme cases you may want to use these options to manage the amount of information generated by the `-debugmsg +relay` option.)

```
"File" = "/blanco"
```

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

```
"SpyExclude" = "WM_SIZE;WM_TIMER;"
```

Excludes debug messages about WM_SIZE and WM_TIMER in the logfile.

```
"SpyInclude" = "WM_SIZE;WM_TIMER;"
```

Includes debug messages about WM_SIZE and WM_TIMER in the logfile.

```
"RelayInclude" = "user32.CreateWindowA;comctl32.*"
```

Include only the listed functions in a *-debugmsg +relay* trace. This entry is ignored if there is a *RelayExclude* entry.

```
"RelayExclude" = "RtlEnterCriticalSection;RtlLeaveCriticalSection"
```

Exclude the listed functions in a *-debugmsg +relay* trace. This entry overrides any settings in a *RelayInclude* entry. If neither entry is present then the trace includes everything.

In both entries the functions may be specified either as a function name or as a module and function. In this latter case specify an asterisk for the function name to include/exclude all functions in the module.

[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 system (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 [Network] Section

[Network] contains settings related to networking. Currently there is only one value that can be set.

UseDnsComputerName

A boolean setting (default: Y) that affects the way Wine sets the computer name. The computer name in the Windows world is the so-called *NetBIOS name*. It is contained in the `ComputerName` in the registry entry `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ComputerName\ComputerName`.

If this option is set to "Y" or missing, Wine will set the NetBIOS name to the Unix host name of your computer, if necessary truncated to 31 characters. The Unix hostname is the output of the shell command **hostname**, up to but not including the first dot ('.'). Among other things, this means that Windows programs running under Wine cannot change the NetBIOS computer name.

If this option is set to "N", Wine will use the registry value above to set the NetBIOS name. Only if the registry entry doesn't exist (usually only during the first wine startup) it will use the Unix hostname as usual. Windows applications can change the NetBIOS name. The change will be effective after a "reboot", i.e. after restarting Wine.

The [AppDefaults] Section

The section is used to overwrite certain settings 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 corresponding "standard" section of the configuration file that should have some of its settings overwritten with the application specific settings you define. The three parts of the section name are separated by two backslashes.

Currently wine supports overriding selected settings within the sections [DllOverrides], [x11drv], [version] and [dsound] only.

Here is an example that overrides 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 (which only is used for your user).

So copy your version of the wine config file to `$HOME/.wine/config` or `/usr/local/etc/wine.conf` 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 IRC channel `#WineHQ` found on `irc.openprojects.net`, or connected servers. Make sure that you have looked over this document thoroughly, and have also read:

- `README`
- <http://www.winehq.org/trouble/>

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

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 have 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

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.

Managed

The default. Specified by using the `Managed` wine config file 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 currently being done, though, and it's planned to be finished before the Wine 1.0 release).

Unmanaged/Normal

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. Unmanaged mode is only used if both `Managed` and `Desktop` are set to disabled.

Desktop-in-a-Box

Specified by using the `Desktop` wine config file option (see below). (adding a geometry, e.g. `800x600` for a such-sized desktop, or even `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. Note: currently there's one desktop window for every application; this will be fixed at some time.

The [x11drv] section

Managed

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

Desktop

Creates a main desktop window of a specified size to display all Windows applications in. The size argument could e.g. be `"800x600"`.

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).

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.

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.

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.

Synchronous

To be used for debugging X11 operations. If Wine crashes with an X11 error, then you should enable Synchronous mode to disable X11 request caching in order to make sure that the X11 error happens directly after the corresponding X11 call in the log file appears. Will slow down X11 output !

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 the DISPLAY environment variable.

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.

TextCP

Codepage to be used for rendering the text in X11 output. Some sample values would be 437 (USA, Canada), 850 (Europe), 852 (Central/Eastern Europe), 855 (Cyrillic). For additional suitable values, see e.g. the Linux kernel's codepage configuration page.

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 it's stored in the hidden file `system.dat`) contains everything pertaining to the current Windows installation.

HKEY_USERS

This fundamental root key (in win9x it's 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 want (make sure you don't have Wine running when modifying them, otherwise your changes will be discarded).

FIXME: global config currently not implemented. 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.

GlobalRegistryDir

Optional. Sets the path to look for the Global 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.)

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.

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.

UseNewFormat

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

Setting the windows and DOS version value that's passed to programs

Written by Andreas Mohr <amohr@codeweavers.com> Oct 18 2002

The windows and DOS version value a program gets e.g. by calling the Windows function `GetVersion()` plays a very important role: If your Wine installation for whatever reason fails to provide to your program the correct version value that it expects, then the program might assume some very bad things and fail (in the worst case even silently !). Fortunately Wine contains some more or less intelligent Windows version guessing algorithm that will try to guess the Windows version a program might expect and pass that one on to the program. Thus you should *not* lightly configure a version value, as this will be a "forced" value and thus turn out to be rather harmful to proper operation. In other words: only explicitly set a Windows version value in case Wine's own version detection was unable to provide the correct Windows version and the program fails.

How to configure the Windows and DOS version value Wine should return

The version values can be configured in the wine config file in the [Version] section.

"Windows" = "<version string>"

default: none; chosen by semi-intelligent detection mechanism based on DLL environment. Used to specify which Windows version to return to programs (forced value, overrides standard detection mechanism !). Valid settings are e.g. "win31", "win95", "win98", "win2k", "winxp". Also valid as an AppDefaults setting (recommended/preferred use).

"DOS" = "<version string>"

Used to specify the DOS version that should be returned to programs. Only takes effect in case Wine acts as "win31" Windows version ! Common DOS version settings include 6.22, 6.20, 6.00, 5.00, 4.00, 3.30, 3.10. Also valid as an AppDefaults setting (recommended/preferred use).

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

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

How To Set Up?

Reading labels and serial numbers just works automatically 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 config file 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.

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/config` or `wine.conf`, 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 explicitly 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):

`kernel386, 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`.

`w32skrn1`

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`.

crt.dll

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.driv

MIDI Sequencer MCI driver (.MID playback).

mcwave.driv

Wave audio MCI driver (.WAV playback).

mciavi.driv

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

mcianim.driv

Animation MCI driver.

msacm.driv

Audio Compression Manager. Set to same as msacm32.

midimap.driv

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:

1. The directory the program was started from.
2. The current directory.
3. The Windows system directory.
4. The Windows directory.
5. 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. Note that you probably shouldn't use NT-based native DLLs, since Wine's NT API support is somewhat weaker than its Win9x API support (thus leading to even worse compatibility with NT DLLs than with a no-windows setup !), so better use Win9x native DLLs instead or no native DLLs at all.

Fetching native DLLs from a Windows CD

Written by Andreas Mohr <amohr@codeweavers.com>

The Linux **cabextract** utility can be used to extract native Windows .dll files from .cab files that are to be found on many Windows installation CDs.

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 **bdf2pcf**.
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/xfst 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/Xf86ttt`. And there is **xfstt** which uses the freetype library, see freetype description).

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 $\{0, \dots, N\}$ 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"
-...	->	
-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 `xfonstest` 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 a freetype source archive (`freetype-X.Y.tar.gz` ?).
2. Read docs, unpack, configure and install
3. Test the library, e.g. `ftview 20 /dos/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 `xf`s 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 the wine config file. 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 the wine config file 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 the wine config file, 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>

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
```

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 1-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.

Using ODBC

This section describes how ODBC works within Wine and how to configure it to do what you want (if it can do what you want).

The ODBC system within wine, as with the printing system, is designed to hook across to the Unix system at a high level. Rather than ensuring that all the windows code works under wine it uses a suitable Unix ODBC provider, such as unixODBC. Thus if you configure Wine to use the builtin odbc32.dll that wine dll will interface to your Unix ODBC package and let that do the work, whereas if you configure Wine to use the native odbc32.dll it will try to use the native ODBC32 drivers etc.

Using a Unix ODBC system with Wine

The first step in using a Unix ODBC system with Wine is, of course, to get the Unix ODBC system working itself. This may involve downloading code or rpms etc. There are several Unix ODBC systems available; the one the author is used to is unixODBC (with the IBM DB2 driver). Typically such systems will include a tool, such as isql, which will allow you to access the data from the command line so that you can check that the system is working.

The next step is to hook the Unix ODBC library to the wine builtin odbc32 dll. The builtin odbc32 (currently) looks to the environmental variable `LIB_ODBC_DRIVER_MANAGER` for the name of the odbc library. For example in the author's .bashrc file is the line:

```
export LIB_ODBC_DRIVER_MANAGER=/usr/lib/libodbc.so.1.0.0
```

If that environmental variable is not set then it looks for a library called libodbc.so and so you can add a symbolic link to equate that to your own library. For example as root you could run the commands:

```
ln -s libodbc.so.1.0.0 /usr/lib/libodbc.so
/sbin/ldconfig
```

The last step in configuring this is to ensure that Wine is set up to run the builtin version of odbc32.dll, by modifying the DLL configuration. This builtin dll merely acts as a stub between the calling code and the Unix ODBC library.

If you have any problems then you can use the debugmsg channel odbc32 to trace what is happening. One word of warning. Some programs actually cheat a little and bypass the odbc library. For example the Crystal Reports engine goes to the registry to check on the DSN. The fix for this is documented at unixODBC's site where there is a section on using unixODBC with Wine.

Using Windows ODBC drivers

Does anyone actually have any experience of this and anything to add?

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
-help, -h      Show this help message
-version, -v    Display the Wine version
```

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
```

(note the backslash-escaped "\\")

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 your program directly in the Unix console where you started it, 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, a lot of programs 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. (As described in the Debug section of configuring wine you can modify what the *relay* trace reports). 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 will 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	imagehelp
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
mmax	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

Specifies whether to load the builtin or the native (if available) version of a DLL. Example:

```
$ wine -dll setupx=n foo.exe
```

See the DLL chapter for more details.

-help

Shows a small command line help page.

-version

Shows the Wine version string. Useful to verify your installation.

Setting Windows/DOS environment variables

Your program might require some environment variable to be set properly in order to run successfully. In this case you need to set this environment variable in the Linux shell, since Wine will pass on the entire shell environment variable settings to the Windows environment variable space. Example for the bash shell (other shells may have a different syntax !):

```
export MYENVIRONMENTVAR=myenvironmentvarsetting
```

This will make sure your Windows program can access the MYENVIRONMENTVAR environment variable once you start your program using Wine. If you want to have MYENVIRONMENTVAR set permanently, then you can place the setting into `/etc/profile`, or also `~/.bashrc` in the case of bash.

Note however that there is an exception to the rule: If you want to change the PATH environment variable, then of course you can't modify it that way, since this will alter the Unix PATH environment setting. Instead, you should set the WINEPATH environment variable. An alternative way to indicate the content of the DOS PATH environment variable would be to change the "path" setting in the wine config file's [wine] section.

Chapter 6. Troubleshooting / Reporting bugs

What to do if some program still doesn't work ?

There are times when you've been trying everything, you even killed a cat at full moon and ate it with rotten garlic and foul fish while doing the Devil's Dance, yet nothing helped to make some damn program work on some Wine version. Don't despair, we're here to help you... (in other words: how much do you want to pay ?)

Run "winecheck" to check your configuration

Run a Perl script called **winecheck**, to be found in Wine's tools/ directory. The latest version can always be found at <http://home.arcor.de/andi.mohr/download/winecheck>. Make sure to run **chmod +x winecheck** first before trying to execute it... (or alternatively run it via **perl ./winecheck**) The winecheck output will be a percentage score indicating Wine configuration correctness. Note that winecheck is only alpha, so it's not very complete or 100% accurate.

Use different windows version settings

In several cases using different windows version settings can help.

Use different startup paths

This sometimes helps, too: Try to use both **wine prg.exe** and **wine x:\\full\\path\\to\\prg.exe**

Fiddle with DLL configuration

Run with **-debugmsg +loaddll** to figure out which DLLs are being used, and whether they're being loaded as native or builtin. Then make sure you have proper native DLL files in your configured C:\\windows\\system directory and fiddle with DLL load order settings at command line or in config file.

Check your system environment !

Just an idea: could it be that your Wine build/execution environment is broken ? Make sure that there are no problems whatsoever with the packages that Wine depends on (gcc, glibc, X libraries, OpenGL (!), ...) E.g. some people have strange failures to find stuff when using "wrong" header files for the "right" libraries !!! (which results in days of debugging to desperately try to find out why that lowlevel function fails in a way that is completely beyond imagination... ARGH !)

Use different GUI (Window Manager) modes

Instruct Wine via config file to use either desktop mode, managed mode or plain ugly "normal" mode. That can make one hell of a difference, too.

Check your app !

Maybe your app is using some kind of copy protection ? Many copy protections currently don't work on Wine. Some might work in the future, though. (the CD-ROM layer isn't really full-featured yet).

Go to GameCopyWorld (<http://www.gamecopyworld.com>) and try to find a decent crack for your game that gets rid of that ugly copy protection. I hope you do have a legal copy of the program, though... :-)

Check your Wine environment !

Running with or without a Windows partition can have a dramatic impact. Configure Wine to do the opposite of what you used to have. Also, install DCOM98 or DCOM95. This can be very beneficial.

Reconfigure Wine

Sometimes wine installation process changes and new versions of Wine account on these changes. This is especially true if your setup was created long time ago. Rename your existing `~/ .wine` directory for backup purposes. Use the setup process that's recommended for your Wine distribution to create new configuration. Use information in old `~/ .wine` directory as a reference. For source wine distribution to configure Wine run `tools/wineinstall` script as a user you want to do the configuration for. This is a pretty safe operation. Later you can remove the new `~/ .wine` directory and rename your old one back.

Check out further information

Check out the Wine Troubleshooting Guide (<http://www.winehq.org/fom-meta/cache/19.html>) on WineHQ. Go to Google Groups (<http://groups.google.com>) and check whether some guys are smarter than you ;-) (well, whether they found a solution to the problem, that is) Go to WineHQ's Application Database (<http://appdb.codeweavers.com>) and check whether someone posted the vital config hint for your app. If that doesn't help, then consider going to `irc.openprojects.net` channel `#WineHQ`, posting to `news:comp.emulators.ms-windows.wine` or mailing to the `wine-users` (or maybe sometimes even `wine-devel`) mailing lists.

Debug it!

Have you used the Search feature of the Wine Troubleshooting Guide (<http://www.winehq.org/fom-meta/cache/19.html>) ?? (i.e. are you sure there's no answer ?) If you have, then try The Perfect Enduser Wine Debugging Guide (<http://www.winehq.org/fom-meta/cache/230.html>), and of course don't forget to read the Wine Developers Guide.

How To Report A Bug

Please report all bugs along any relevant information to Wine Bugzilla (<http://bugs.winehq.com/>). Please, search the Bugzilla database to check whether your problem is already reported. If it is already reported please add any relevant information to the original bug report.

All Bug Reports

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

1. Post as much relevant 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:

- Which version of Wine you're using (run **wine -v**)
- The name of the Operating system you're using, what distribution (if any), and what version. (i.e., Linux RedHat 7.2)
- Which compiler and version, (run **gcc -v**). If you didn't compile wine then the name of the package and where you got it from.
- Windows version, if used with Wine. Mention if you don't use Windows.
- The name of the program you're trying to run, its version number, and a URL for where the program can be obtained (if available).
- The exact command line you used to start wine. (i.e., **wine "C:\Program Files\Test\program.exe"**).
- The exact steps required to reproduce the bug.
- 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**).

This will output additional information at the console that may be helpful in debugging the program. It also slows the execution of program. There are some cases where the bug seems to disappear when `+relay` is used. Please mention that in the bug report.

Crashes

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 one of the following methods:

The Easy Way

1. This method is meant to allow even a total novice to submit a relevant trace log in the event of a crash.

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

It is likely that only the last 100 or so lines of the trace are necessary to find out where the program crashes. In order to get those last 100 lines we need to do the following

1. Redirect all the output of `-debugmsg` to a file.
2. Separate the last 100 lines to another file using `tail`.

This can be done using one of the following methods.

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 cycles.)

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.

Post the bug to Wine Bugzilla (<http://bugs.winehq.com/>). You need to attach the output file `report_file` from part 2). Along with the the relevant information used to create it. Do not cut and paste the report in the bug description - it is pretty big and it will make a mess of the bug report. If you do this, your chances of receiving some sort of helpful response should be very good.

Please, search the Bugzilla database to check whether your problem is already reported. If it is already reported attach the output file `report_file` to the original bug report and add any other relevant information.

