Wine User Guide

Wine User Guide

Table of Contents

Chapter 1. Introduction

Overview / About

Purpose of this document and intended audience

This document, called the Wine User Guide, is both an easy installation guide and an extensive reference guide. This guide is for both the new Wine user and the experienced Wine user, offering full step-by-step installation and configuration instructions, as well as featuring extensive reference material by documenting all configuration features and support areas.

Further questions and comments

If, after examining this guide, the FAQ, and other relevant documentation there is still something you cannot figure out, we would love to hear from you. The mailing lists (http://www.winehq.org/site/forums) section contains several mailing lists and an IRC channel, all of which are great places to seek help and offer suggestions. If you are particularly savvy, and believe that something can be explained better, you can file a bug report (http://bugs.winehq.org/) or post a patch (http://www.winehq.org/site/sending_patches) on Wine's documentation itself.

Content overview / Steps to take

In order to be able to use Wine, you must first have a working installation. This guide will help you to move your system from an empty, Wineless void to one boasting a fresh, up to date Wine install. The first step, Getting Wine, illustrates the various methods of getting Wine's files onto your computer. The second step, Installing Wine, details the various install processes available to you. The third step, Configuring Wine, shows how to customize a Wine installation depending on your individual needs. The final step, Running Wine, covers the specific steps you can take to get a particular application to run better under Wine, and provides useful links in case you need further help.

What is Wine?

Windows and Linux

Different software programs are designed for different operating systems, and most won't work on systems that they weren't designed for. Windows programs, for example, won't run in Linux because they contain instructions that the system can't understand until they're translated by the Windows environment. Linux programs, likewise, won't run under the Windows operating system because Windows is unable to interpret all of their instructions.

This situation presents a fundamental problem for anyone who wants to run software for both Windows and Linux. A common solution to this problem is to install both operating systems on the same computer, known as "dual booting." When a Windows program is needed, the user boots the machine into Windows to run it; when a Linux program is then needed, the user then reboots the machine into Linux. This option presents great difficulty: not only must the user endure the frustration of frequent rebooting, but programs for both platforms can't be run simultaneously. Having Windows on a system also creates an added burden: the software is expensive, requires a separate disk partition, and is unable to read most filesystem formats, making the sharing of data between operating systems difficult.

What is Wine, and how can it help me?

Wine makes it possible to run Windows programs alongside any Unix-like operating system, particularly Linux. At its heart, Wine is an implementation of the Windows Application Programing Interface (API) library, acting as a bridge between the Windows program and Linux. Think of Wine as a compatibility layer, when a Windows program tries to perform a function that Linux doesn't normally understand, Wine will translate that program's instruction into one supported by the system. For example, if a program asks the system to create a Windows pushbutton or text-edit field, Wine will convert that instruction into its Linux equivalent in the form of a command to the window manager using the standard X11 protocol.

If you have access to the Windows program's source code, Wine can also be used to recompile a program into a format that Linux can understand more easily. Wine is still needed to launch the program in its recompiled form, however there are many advantages to compiling a Windows program natively within Linux. For more information, see the Winelib User Guide.

Wine features

Throughout the course of its development, Wine has continually grown in the features it carries and the programs it can run. A partial list of these features follows:

- Support for running Win32 (Win 95/98, NT/2000/XP), Win16 (Win 3.1) and DOS programs
- Optional use of external vendor DLL files (such as those included with Windows)
- X11-based graphics display, allowing remote display to any X terminal, as well as a text mode console
- Desktop-in-a-box or mixable windows
- DirectX support for games
- Good support for various sound drivers including OSS and ALSA
- Support for alternative input devices
- Printing: PostScript interface driver (psdrv) to standard Unix PostScript print services
- Modem, serial device support
- Winsock TCP/IP networking support
- ASPI interface (SCSI) support for scanners, CD writers, and other devices
- Advanced unicode and foreign language support
- Full-featured Wine debugger and configurable trace logging messages for easier troubleshooting

Versions of Wine

Wine from Wine HQ

Wine is an open source project, and there are accordingly many different versions of Wine for you to choose from. The standard version of Wine comes in intermittant releases (roughly once a month), and can be downloaded over the internet in both prepackaged binary form and ready to compile source code form. Alternatively, you can install a development version of Wine by using the latest available source code on the CVS server. See the next chapter, Getting Wine, for further details.

Other Versions of Wine

There are a number of programs that are derived from the standard Wine codebase in some way or another. Some of these are commercial products from companies that actively contribute to the Wine project.

These products try to stand out or distinguish themselves from the standard version of Wine by offering greater compatibility, easier configuration, and commercial support. If you require such things, it is a good idea to consider purchasing these products.

Table 1-1. Various Wine offerings

Product	Description	Distribution Form
CodeWeavers CrossOver Office (http://www.codeweavers.com/products/office)	CrossOver Office allows you to install your favorite Windows productivity applications in Linux, without needing a Microsoft Operating System license. CrossOver includes an easy to use, single click interface, which makes installing a Windows application simple and fast.	Commercial; 30-day fully-functional demo available.
CodeWeavers CrossOver Office Server Edition (http://www.codeweavers.com/products/cxofficeserver)	CrossOver Office Server Edition allows you to run your favorite Windows productivity applications in a distributed thin-client environment under Linux, without needing Microsoft Operating System licenses for each client machine. CrossOver Office Server Edition allows you to satisfy the needs of literally hundreds of concurrent users, all from a single server.	Commercial
TransGaming Technologies Cedega (http://www.transgaming.com)	Cedega, formerly known as WineX, is a Wine version derived from the old BSD licensed Wine tree. It is designed specifically to function with games, and so development is centered around ensuring compatibility with technologies like Direct3D, DirectX, and various methods of copy protection.	Commercial; free CVS download of reduced version (no copy protection support etc.)
ReWind (http://sourceforge.net/projects/rewind)	ReWind is a Wine version derived from the old BSD licensed Wine tree. Due to its BSD license it can't incorporate most recent Wine patches, however its code can be used by any software project without restriction. Although code is occasionally contributed to ReWind alongside the main Wine project, there has never been an official release of ReWind.	Free, Open Source: BSD license

Alternatives to Wine you might want to consider

There are many ways to run software other than through Wine. If you are considering using Wine to run an application you might want to think about the viability of these approaches if you encounter difficulty.

Native Applications

Instead of running a particular Windows application with Wine, one frequently viable alternative is to simply run a different application. Many Windows applications, particularly more commonly used ones such as media players, instant messengers, and filesharing programs have very good open source equivalents. Furthermore, a sizable number of Windows programs have been ported to Linux directly, eliminating the need for Wine (or Windows) entirely.

Another Operating System

Probably the most obvious method of getting a Windows application to run is to simply run it on Windows. However, security, license cost, backward-compatibility, and machine efficiency issues can make this a difficult proposition, which is why Wine is so useful in the first place.

Another alternative is to use ReactOS (http://www.reactos.com), which is a fully open source alternative to Windows. ReactOS shares code heavily with the Wine project, but rather than running Windows applications on top of Linux they are instead run on top of the ReactOS kernel. ReactOS also offers compatibility with Windows driver files, allowing the use of hardware without functional Linux drivers.

Virtual Machines

Rather than installing an entirely new operating system on your machine, you can instead run a virtual machine at the software level and install a different operating system on it. Thus, you could run a Linux system and at the same time run Windows along with your application in a virtual machine simultaneously on the same hardware. Virtual machines allow you to install and run not only different versions of Windows on the same hardware, but also other operating systems, including ReactOS.

There are several different virtual machine offerings out there, and some are also able to emulate x86 hardware on different platforms. The open source Bochs (http://bochs.sourceforge.net/) and QEMU (http://fabrice.bellard.free.fr/qemu/) can run both Windows and ReactOS virtually. Other, commercial virtual machine offerings include VMware (http://www.vmware.com/) and Microsoft's VirtualPC (http://www.microsoft.com/windowsxp/virtualpc/).

There are significant drawbacks to using virtual machines, however. Unlike Wine, such programs *are* emulators, so there is an inevitable speed decrease which can be quite substantial. Furthermore, running an application inside a virtual machine prevents fully integrating the application within the current environment. You won't, for example, be able to have windows system tray icons or program shortcuts sitting alongside your desktop Linux ones, since instead the Windows applications must reside completely within the virtual machine.

Chapter 2. Getting Wine

If you decided that you can use and want to use Wine (e.g. after having read the introductory chapter), then as a first step you need to find a good compatible Wine version that you like and that works on your system, and after you found one, the next step is to transfer its files to your system somehow. This chapter is here to tell you what you need to take care of in order to successfully accomplish these two steps.

How to download Wine?

There are three different methods of how the files belonging to Wine may be brought (downloaded) to your system:

- Getting a single Wine *package* file (specifically adapted to your particular system), which contains various *binary* files of Wine
- Getting a single compressed archive file (usually .tar.gz), which contains all *source code* files of a standard Wine release version
- Downloading from a *CVS* server, which contains the very latest development source code files of Wine

Which Wine form should I pick?

Now that we told you about the different Wine distribution methods available, let's discuss the advantages and disadvantages of the various methods.

Wine distribution methods

Wine package file

Intended user level: Beginner to Advanced

Using Wine package files is easy for three reasons: They install everything else that's needed for their operation, they usually preconfigure a lot, and you don't need to worry about compiling anything or so. You can get the official wine packages from the sourceforge.net Wine download page (http://sourceforge.net/project/showfiles.php?group_id=6241)

Wine source code via archive file

Intended user level: Advanced to Expert

A Wine source code archive file can be used if you want to compile your own standard Wine release. By using differential patch files to newer Wine versions, you can easily upgrade your outdated Wine directory. However, as you need to manually download patch files and you're only able to download the most current standard Wine release, this is not necessarily the best method to use. The only advantage a Wine source archive has is that it is a standard Wine release with less development "quirks" than current CVS code. Except for that, CVS source code is much preferred and almost as easy.

Wine source code via CVS checkout

Intended user level: Advanced to Expert/Developer

The Wine CVS checkout offers the best way to take part in bleeding edge Wine capabilities and development, since you'll be able to download every single CVS commit even *beyond* the last official Wine release. As upgrading a Wine CVS checkout tree to the latest version is very easy, this is a recommended method of installing Wine. Plus, by carefully following the instructions in this Guide, you'll be able to gain the very best Wine environment compatibility.

If you are running a distribution of Linux or some other system 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 system packages between different distributions, and even from different versions of the same distribution. Often a package will only work on the distribution which it has been compiled for. We'll cover Debian Linux, Red Hat, Mandrake, SUSE and Slackware Linux, FreeBSD, 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 need 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 a Wine package

Debian Linux

In most cases on a Debian system (or any other distribution that uses packages that use the file name ending .deb, for that matter), you can download and 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. You might first need to properly update your package setup, though, by using an *editor* as *root* to add an entry to `/etc/apt/sources.list` to point to an active package server and then running **apt-get update**.

Once you're done with that step, you may skip the Wine installation chapter, since apt-get has not only downloaded, but also installed the Wine files already. Thus you can now go directly to the Configuration section.

However, if you don't want to or cannot use the automatic download method for .deb packages that **apt-get** provides, then please read on.

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 so-called "unstable" Debian 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).

If you downloaded a separate .deb package file (e.g. a newer Wine release as stated above) that's not part of your distribution and thus cannot be installed via **apt-get**, you must use **dpkg** instead. For instructions on how to do this, please proceed to the Installation section.

Linux Red Hat, Mandrake, SUSE, and Slackware

Red Hat, Mandrake, SUSE and Slackware users can download a wine binary from the sourceforge.net Wine download page (http://sourceforge.net/project/showfiles.php?group_id=6241)

FreeBSD

In order to use Wine you need to build and install a new kernel with options USER_LDT, SYSVSHM, SYSVSEM, and SYSVMSG.

If you want to install Wine using the FreeBSD port system, run in a *terminal*:

```
$ su -
# cd /usr/ports/emulators/wine/
# make
# make install
# make clean
```

This process will get wine source from the Internet, then download the Wine package and install it on your system.

If you want to install Wine from the FreeBSD CD-ROM, run in a *terminal*:

```
$ su -
# mount /cdrom
# cd /cdrom/packages/All
# pkg_add wine_.X.X.X.tgz
```

These FreeBSD install instructions completely install the Wine files on your system; you may then proceed to the Configuration section.

You can also download a FreeBSD package of wine from the sourceforge.net Wine download page (http://sourceforge.net/project/showfiles.php?group_id=6241)

Other systems

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

You could also try to use Google (http://www.google.com/search?q=wine+package+download) to track down miscellaneous distribution packages.

Getting Wine source code

If you are going to compile Wine (instead of installing binary Wine files), either to use the most recent code possible or to improve it, then the first thing to do is to obtain a copy of the source code. We'll cover how to retrieve and compile the source releases from the official archives, and also how to get the cutting edge up-to-the-minute fresh Wine source code from CVS (Concurrent Versions System).

Once you have downloaded Wine source code according to the instructions below, there are two ways to proceed: If you want to manually install and configure Wine, then go to the Compiling section. If instead you want automatic installation, then go straight to the Configuration section to make use of **wineinstall** to automatically install and configure Wine.

You may also need to know how to apply a source code patch to your version of Wine. Perhaps you've uncovered a bug in Wine, reported it to the Wine Bugzilla (http://bugs.winehq.org) or the Wine mailing list (mailto:wine-devel@winehq.org), and received a patch from a developer to hopefully fix the bug. We will show you how to safely apply the patch and revert it if it doesn't work.

Getting Wine Source Code from the official archives

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

- ftp://ftp.ibiblio.org/pub/Linux/ALPHA/wine/development/ (ftp://ftp.ibiblio.org/pub/Linux/ALPHA/wine/development/)
- sourceforge.net download page (http://sourceforge.net/project/showfiles.php?group_id=6241&package_id=77449)

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

I'd recommend placing the Wine archive file that you chose into the directory where you intend to extract Wine. In this case, let's just assume that it is your home directory.

Once you have downloaded a Wine archive file, we need to extract the archive file. This is not very hard to do. First switch to the directory containing the file you just downloaded. Then extract the source in a *terminal* with (e.g.):

```
$ tar xvzf wine-20030115.tar.gz
```

Just in case you happen to get a Wine archive that uses `.tar.bz2` extension instead of `.tar.gz`: Simply use **tar xvjf** in that case instead.

Since you now have a fully working Wine source tree by having followed the steps above, you're now well-prepared to go to the Wine installation and configuration steps that follow.

Getting Wine Source Code from CVS

This part is intended to be quick and easy, showing the bare minimum of what is needed to download Wine source code via CVS. If you're interested in a very verbose explanation of CVS or advanced CVS topics (configuration settings, CVS mirror servers, other CVS modules on WineHQ, CVSWeb, ...), then please read the full CVS chapter in the Wine Developer's Guide.

CVS installation check

First you need to make sure that you have **cvs** installed. To check whether this is the case, please run in a *terminal*:

```
$ cvs
```

If this was successful, then you should have gotten a nice CVS "Usage" help output. Otherwise (e.g. an error "cvs: command not found") you still need to install a CVS package for your particular operating system, similar to the instructions given in the chapters for getting and installing a Wine package on various systems.

Downloading the Wine CVS tree

Once CVS is installed, you can now do a login on our CVS server and checkout (download) the Wine source code. First, let's do the server login, to connect to the US server:

```
$  export CVSROOT=:pserver:cvs@cvs.winehq.org:/home/wine
$  cvs login
```

To connect to the EU server:

```
$ export CVSROOT=:pserver:cvs@rhlx01.fht-esslingen.de:/home/wine
```

```
$ cvs login
```

If **cvs** successfully connects to the CVS server, then you will get a "CVS password:" prompt. Simply enter "cvs" as the password (the password is *case sensitive*: no capital letters!).

After login, we are able to download the Wine source code tree. Please make sure that you are in the directory that you want to have the Wine source code in (the Wine source code will use the subdirectory `wine/` in this directory, since the subdirectory is named after the CVS module that we want to check out). We assume that your current directory might be your user's home directory. To download the Wine tree into the subdirectory `wine/`, run:

```
$  cvs -z3 checkout wine
```

Downloading the CVS tree might take a while (some minutes to few hours), depending on your connection speed. Once the download is finished, you should keep a note of which directory the newly downloaded `wine/` directory is in, by running **pwd** (Print Working Directory):

```
$ pwd
```

Later, you will be able to change to this directory by running:

```
$ cd <some_dir>
```

where <some_dir> is the directory that **pwd** gave you. By running

```
$ cd wine
```

you can now change to the directory of the Wine CVS tree you just downloaded. Since you now have a fully working Wine source tree by having followed the steps above, you're now well-prepared to go to the Wine installation and configuration steps that follow.

Updating the Wine CVS tree

After a while, you might want to update your Wine CVS tree to the current version. Before updating the Wine tree, it might also be a good idea to run **make uninstall** as root in order to uninstall the installation of the previous Wine version.

To proceed with updating Wine, simply **cd** to the Wine CVS tree directory, then run, if you're using the US server:

```
$ make distclean
$ cvs update -PAd
```

The **make distclean** part is optional, but it's a good idea to remove old build and compile configuration files before updating to a newer Wine version. Once the CVS update is finished, you can proceed with installing Wine again as usual.

Updating Wine with a Patch

If you got Wine source code (e.g. via a tar archive file), you have the option of applying patches to the source tree to update to a newer Wine release or to fix bugs and add experimental features. Perhaps you've found a bug, reported it to the Wine mailing list (mailto:wine-devel@winehq.org), and received a patch file to fix the bug. You can apply the patch with the **patch** command, which takes a streamed patch from `stdin`:

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

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

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

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

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

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

The *-p0* option to **patch** tells it to keep the full file name from the patch file. For example, if the file name in the patch file was `wine/programs/clock/main.c`. Setting the *-p0* option would apply the patch to the file of the same name i.e. `wine/programs/clock/main.c` . Setting the *-p1* option would strip off the first part of the file name and apply the patch to `programs/clock/main.c`. The *-p1* option would be useful if you named your top level wine directory differently than the person who sent you the patch. For the *-p1* option **patch** should be run from the top level wine directory.

Chapter 3. Compiling the Wine Source

In case you downloaded Wine source code files, this chapter will tell you how to compile it into binary files before installing them. Otherwise, please proceed directly to the Installation chapter to install the binary Wine files.

Compiling Wine

Commands

To compile and install Wine, run the following commands:

```
./configure
make depend
make
make install
```

Please note that the last command (**make install**) must be run as root.

Requirements

For an up-to-date list of software requirements for compiling Wine and instructions how to actually do it, please see the README (http://www.winehq.org/source/README) file, which is also available in the main directory of a Wine source code tree.

Space required

You also need about 400 MB of available disk space for compilation. The compiled libwine.so binary takes around 5 MB of disk space, which can be reduced to about 1 MB by stripping ('strip wine'). Stripping is not recommended, however, as you can't submit proper crash reports with a stripped binary.

Common problems

If you get a repeatable sig11 compiling shellord.c, thunk.c or other files, try compiling just that file without optimization (removing the -Ox option from the GCC command in the corresponding Makefile).

Chapter 4. Installing or uninstalling Wine

A standard Wine distribution form (which you probably downloaded according to chapter Getting Wine) includes quite a few different programs, libraries and configuration files. All of these must be set up properly for Wine to work well. In order to achieve this, this chapter will guide you through the necessary steps to get the Wine files installed on your system. It will *not* deal with how to get Wine's Windows environment *configured*; that's what the next chapter will talk about.

When installing Wine, you should make sure that it doesn't happen to overwrite a previous Wine installation (as this would cause an overwhelming amount of annoying and fatal conflicts); uninstalling any previous Wine version (as explained in this chapter) to avoid this problem is recommended.

Installing or uninstalling Wine packages

Now that you have downloaded the Debian or RPM or whatever Wine package file, probably via the instructions given in the previous chapter, you may be wondering "What in the world do I do with this thing?". This section will hopefully be able to put an end to your bewildered questioning, by giving detailed install instructions for all sorts of well-known package types.

Debian Linux

In case you haven't downloaded and automatically installed the Wine package file via **apt-get** as described in the Getting Wine section, you now need to use **dpkg** to install it. Switch to the directory you downloaded the Debian .deb package file to. Once there, type these commands, adapting the package file name as required:

```
$ su -
Password:
# cd /home/user
# dpkg -i wine_0.0.20030115-1.deb
```

(Type the root password at the "Password:" prompt)

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.

Uninstalling an installed Wine Debian package can be done by running:

```
# dpkg -l|grep wine
```

The second column of the output (if any) of this command will indicate the installed packages dealing with "wine". The corresponding packages can be uninstalled by running:

```
# dpkg -r <package_name>
```

where <package_name> is the name of the Wine-related package which you want to uninstall.

Linux Red Hat, Mandrake, SUSE and other distributions using RPM

Most distributions provide a graphical tool for installing RPM packages, you can use it by simply clicking (Or double clicking, depending on your system settings) on the RPM. If you don't have a graphical RPM manager installed, using a shell, switch to the directory where you downloaded the RPM package file to. Once there, type this one command as root, adapting the package file name as required:

```
# rpm -ivh wine-20031212.i386.rpm
```

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

If you've installed wine graphically, you can uninstall it using your graphical RPM manager (Gnorpm, Kpackage, Yast, Mandrake Control Center and so on), alternatively, uninstalling a installed Wine RPM package can be done from a shell, by running:

```
# rpm -qa|grep -i wine
```

This command will indicate the installed packages dealing with "wine". The corresponding packages can be uninstalled by running:

```
# rpm -e <package_name>
```

where <package_name> is the name of the Wine-related package which you want to uninstall.

Installing or uninstalling a Wine source code tree

If you are in the directory of the Wine version that you just compiled (e.g. by having run **make depend && make**), then you may now install this Wine version by running as *root*:

```
# make install
```

This will copy the Wine binary files to their final destination in your system. You can then proceed to the Configuration chapter to configure the Wine environment.

If instead you want to uninstall the currently installed Wine source code version, then change to the main directory of this version and run as *root*:

```
# make uninstall
```

Chapter 5. Configuring Wine

Now that you hopefully managed to successfully install the Wine program files, this chapter will tell you how to configure the Wine environment properly to run your Windows programs.

First, we'll give you an overview about which kinds of configuration and program execution aspects a fully configured Windows environment has to fulfill in order to ensure that many Windows programs run successfully without encountering any misconfigured or missing items. Next, we'll show you which easy helper programs exist to enable even novice users to complete the Wine environment configuration in a fast and easy way. The next section will explain the purpose of the Wine configuration file, and we'll list all of its settings. After that, the next section will detail the most important and unfortunately most difficult configuration part: how to configure the file system and DOS drive environment that Windows programs need. In the last step we'll tell you how to establish a working Windows registry base. Finally, the remaining parts of this chapter contain descriptions of specific Wine configuration items that might also be of interest to you.

What are the requirements of a fully working Windows environment?

A Windows installation is a very complex structure. It consists of many different parts with very different functionality. We'll try to outline the most important aspects of it.

- 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`) directory. Some Windows programs 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 program asks for a DLL, Wine does not simulate the presence of non-existent 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.

Easy configuration helper programs

Managing the Wine configuration file settings can be a difficult task, sometimes too difficult for some people. That's why there are some helper applications for easily setting up an initial wine configuration file with useful default settings.

wineinstall

wineinstall is a small configuration tool residing as `tools/wineinstall` in a Wine source code tree. It has been written to allow for an easy and complete compilation/installation of Wine source code for people who don't bother with reading heaps of very valuable and informative documentation ;-)

Once you have successfully extracted the Wine source code tree, change to the main directory of it and then run (as user):

```
$ ./tools/wineinstall
```

Doing so will compile Wine, install Wine and configure the Wine environment (either by providing access to a Windows partition or by creating a properly configured no-windows directory environment).

Verification of correct configuration

TODO: After you have finished configuring Wine you can verify your Wine configuration by running winecfg. This functionality will be added to winecfg in the near future.

Please check out the configuration documentation below to find out more about Wine's configuration, or proceed to the Troubleshooting chapter.

The Wine Configuration File

This section is meant to contain both an easy step-by-step introduction to the Wine configuration file (for new Wine users) and a complete reference to all Wine configuration file settings (for advanced users).

Configuration File Introduction

The Wine configuration file is the central file to store configuration settings for Wine. This file (which is called `config`) can be found in the sub directory `.wine/` of your user's home directory (directory `/home/user/`). In other words, the Wine configuration file is `~/.wine/config`. Note that since the Wine configuration file is a part of the Wine registry file system, this file also *requires* a correct "WINE REGISTRY Version 2" header line to be recognized properly, just like all other Wine registry text files (just in case you decided to write your own registry file from scratch and wonder why Wine keeps rejecting it).

The settings available in the configuration file include:

- Directory settings
- Port settings
- The Wine look and feel
- Wine's DLL usage
- Wine's multimedia drivers and DLL configuration

Creating Or Modifying The Configuration File

If you just installed Wine for the first time and want to finish Wine installation by configuring it now, then you could use our sample configuration file `config` (which can be found in the directory `documentation/samples/` of the Wine source code directory) as a base for adapting the Wine configuration file to the settings you want. First, I should mention that you should not forget to make sure that any previous configuration file at `~/.wine/config` has been safely moved out of the way instead of simply overwriting it when you will now copy over the sample configuration file.

If you don't have a pre-existing configuration file and thus need to copy over our sample configuration file to the standard Wine configuration file location, do in a *terminal*:

```
$ mkdir ~/.wine/
$ cp dir_to_wine_source_code/documentation/samples/config ~/.wine/config
```

Otherwise, simply use the already existing configuration file at `~/.wine/config`.

Now you can start adapting the configuration file's settings with an *editor* according to the documentation below. Note that you should *only* change configuration file settings if wineserver is not running (in other words: if your user doesn't have a Wine session running), otherwise Wine won't use them - and even worse, wineserver will overwrite them with the old settings once wineserver quits!!

What Does It Contain?

Let's start by giving an overview of which sections a configuration file may contain, and whether the inclusion of the respective section is *needed* or only *recommended* ("recmd").

Section Name	Needed?	What it Does
[wine]	yes	General settings for Wine
[DllOverrides]	recmd	Overrides defaults for DLL loading
[x11drv]	recmd	Graphics driver settings
[fonts]	yes	Font appearance and recognition
[ppdev]	no	Parallelport emulation
[spooler]	no	Print spooling
[ports]	no	Direct port access
[Debug]	no	What to do with certain debug messages
[Registry]	no	Specifies locations of windows registry files
[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

Now let's explain the configuration file sections in a detailed way.

The [wine] Section

The [wine] section of the configuration file contains basic settings for Wine.

```
"Windows" = "c:\\windows"
"ShowDirSymlinks" = "1"
"ShowDotFiles" = "1"
```

For a detailed description of drive layer configuration and the meaning of these parameters, please look at the Disc Drives, Serial and Parallel Ports section.

```
"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 a lot of Windows GUI programs (ttydrv is still pretty "broken" at running graphical apps). 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. Another way to run Wine without display would be to run X11 via Xvnc, then connect to that VNC display using xvncviewer (that way you're still able to connect to your app and configure it if need be).

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

Tells wine whether to allow printing via printer drivers to work. This option isn't needed for our built-in 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 configuration 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.

```
"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.

The [DllOverrides] Section

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

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

For example, to load built-in KERNEL pair (case doesn't matter here):

```
"kernel,kernel32" = "builtin"
```

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

```
"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 programs 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 , 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 information check out the Fonts chapter.

The [spooler] and [ports] Sections

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], 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 *`WINEDEBUG=+relay`*)

```
"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 *`WINEDEBUG=+relay`* trace. This entry is ignored if there is a *`RelayExclude`* entry.

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

Exclude the listed functions in a *`WINEDEBUG=+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.

[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 programs 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 program 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"
```

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.org/), try the newsgroup comp.emulators.ms-windows.wine, or the IRC channel #WineHQ found on irc.freenode.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.

Disc Drives, Serial and Parallel Ports

Extremely Important Prerequisites

If you're planning to include access to a CD-ROM drive in your Wine configuration on Linux, then *make sure* to add the "unhide" mount option to the CD-ROM file system entry in `/etc/fstab`, e.g.:

```
/dev/cdrom /cdrom  iso9660 ro,noauto,users,unhide 0 0
```

Several Windows program setup CD-ROMs or other CD-ROMs chose to do such braindamaged things as marking very important setup helper files on the CD-ROM as "hidden". That's no problem on Windows, since the Windows CD-ROM driver by default displays even files that are supposed to be "hidden". But on Linux, which chose to *hide* "hidden" files on CD by default, this is *FATAL*! (the programs will simply abort with an "installation file not found" or similar error) Thus you should never forget to add this setting.

Short Introduction

Windows applications refer to disc drives by letters such as `A:`, `B:` and `C:`, and to serial and parallel ports by names such as `COM1`: and `LPT1:`.

You need to tell Wine how to interpret these. You do so by specifying the Unix file system nodes and devices that Wine should map them onto, as described later in this section.

You can map a Windows fixed disc drive onto any node in your Unix file system - this need not be the root node of a drive. For example, you could map your Windows drive `C:` onto your Unix directory `/usr/share/wine-C`. Then the Windows folder `C:\Windows\Fonts` would be at `/usr/share/wine-C/Windows/Fonts` in your Unix file system.

Make sure that you have assigned drive letters for directories that will cover all the items Wine needs to access. These include the programs that you run, the data files they need and the Wine debugger (in case anything goes wrong).

It is best to use a number of drive letters, and map them onto directories that cover small sections of the file system containing the files that Wine will need to access. This is safer than simply assigning a single drive letter to the Unix root directory /, which would allow Windows applications to access the whole of your Unix file system (subject, of course, to Unix permissions). If one of them misbehaved, or if you accidentally installed a virus, this might leave you vulnerable.

For replaceable media, such as floppy discs and CD-ROMs, you should map Windows drive letters onto the mount points for these drives in your Unix file system - for example `/mnt/floppy` or `/mnt/cdrom`.

If your applications access serial and parallel ports directly, you should map these onto the corresponding Unix devices - for example `/dev/ttyS0` and `/dev/lp0`.

Windows Directory Structure

Here's the fundamental layout that Windows programs and installers expect and that we thus need to configure properly in Wine. Without it, they seldomly operate correctly. If you intend to use a no-windows environment (not using an existing Windows partition), then it is recommended to use either **WineSetupTk**'s or **wineinstall**'s capabilities to create an initial windows directory tree, since creating a directory structure manually is tiresome and error-prone.

```
C:\                 Root directory of primary disk drive
  Windows\          Windows directory, containing .INI files,
                      accessories, etc.
    System\         Win3.x/95/98/ME directory for common DLLs
                      WinNT/2000 directory for common 16-bit DLLs
    System32\       WinNT/2000 directory for common 32-bit DLLs
    Start Menu\     Program launcher directory structure
      Programs\     Program launcher links (.LNK files) to programs
  Program Files\    Application binaries (.EXE and .DLL files)
```

The dosdevices Directory

The `dosdevices` directory contains the entries that tell Wine how to map Windows disc drive letters onto Unix file system nodes, and how to map Windows serial and parallel ports onto Unix devices. It is located in the `.wine` sub-directory of your home directory, i.e. `~/.wine/dosdevices`.

The entries in the `dosdevices` directory are symbolic links to Unix file system nodes and devices. You can create them by using the **ln** command in a Unix terminal. Alternatively, many File Managers have the capability of creating symbolic links.

For example, if you have decided to map your Windows `C:` drive onto `/usr/share/wine-c`, you could type the following (after changing to your `dosdevices` directory):

```
ln -s /usr/share/wine-c c:
```

Replaceable media are a little more complicated. In addition to creating a link for the file system on the medium, for example:

```
ln -s /mnt/floppy a:
```

you also need to create a link for the device itself. Notice that this has a double colon after the drive letter:

```
ln -s /dev/fd0 a::
```

For serial and parallel ports, you simply create a link to the device; notice that no colon is required after the Windows device name:

```
ln -s /dev/ttyS0 com1
ln -s /dev/lp0 lpt1
```

Windows shares can are mapped into the `unc/` directory so anything trying to access `\\machinename\some\dir\and\file` will look in `~/.wine/dosdevices/unc/machinename/some/dir/and/file`. For example, if you used Samba to mount `\\machinename\some` on `/mnt/smb/machinename/some` then you can do

```
ln -s /mnt/smb/machinename/some unc/machinename/some
```

to make it available in wine (don't forget to create the unc directory if it doesn't alrady exist).

File system settings in the [wine] section

`"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 programs 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, as Windows programs always assume write access to the Windows directory!

`"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. If you *really* need Wine to take into account symlinked directories, then reenable it, but *be prepared for crashes* in certain Windows programs when using the above method! (in other words: enabling it is certainly not recommended)

Old Path, Temp, System configuration are now moved into the registry. See the Environment Variables paragraph.

More detailed explanation about file system differences

Windows uses a different (and inferior) way than Unix to describe the location of files in a computer. Thus Windows programs also expect to find this different way supported by the system. Since we intend to run Windows programs on a Unix system, we're in trouble, as we need to translate between these different file access techniques.

Windows uses drive letters to describe drives or any other form of storage media and to access files on them. For example, common drive names are `C:` for the main Windows system partition on the first harddisk and `A:` for the first floppy drive. Also, Windows uses `\` (backslash) as the directory separator sign, whereas Unix uses `/` (slash). Thus, an example document on the first data partition in Windows might be accessed by the name of `D:\mywork\mydocument.txt`.

So much for the Windows way of doing things.

Well, the problem is, in Unix there is no such thing as "drive letters". Instead, Unix chose to go the much better way of having one single uniform directory tree (starting with the root directory `/`), which has various storage devices such as e.g. harddisk partitions appended at various directory locations within the tree (an example would be `/data1/mywork`, which is the first data partition mounted/attached to a directory called data1 in the root directory `/`; mywork is a sub directory of the data partition file system that's mounted under `/data1`). In Unix, the Windows example document mentioned above could e.g. be accessed by the name of `/data1/mywork/mydocument.txt`, provided that the administrator decided to mount (attach) the first data partition at the directory /data1 inside the Unix directory tree. Note that in Unix, the administrator can *choose* any custom partition location he wants (here, `/data1`), whereas in Windows the system *selects* any drive letter it deems suitable for the first data partition (here, `D:`), and, even worse, if there is some change in partition order, Windows automatically *changes* the drive letter, and you might suddenly find yourself with a first data partition at drive letter `E:`, with all the file naming and referencing confusion that entails. Thus, the Windows way of using ever-changing drive letters is *clearly inferior* to the Unix way of assigning *fixed* directory tree locations for every data storage medium. As we'll see soon, fortunately this Windows limitation of changing drive letters doesn't affect us in Wine at all, since we can properly map *never-changing* drive letters to *fixed* locations inside the Unix directory tree (and even if the location of the respective Unix directory changes, we can still simply update the Wine drive mapping to reflect the updated location and at the same time keep the original drive letter).

OK, now that we know some theory about Windows and Unix drive and filename mapping, it's probably time to ask how Wine achieves the magic of mapping a Unix directory location to a Windows drive...

Wine chose to do the following: In Wine, you don't assign some real physical storage medium (such as a harddisk partition or similar) to each drive letter mapping entry. Instead, you choose certain sub directory trees inside the Unix directory tree (that starts with `/`) that you would like to assign a drive letter to.

Note that for every Unix sub directory tree that you intend to start Windows programs in, it is *absolutely required* to have a Wine drive mapping entry:

For example, if you had a publicly writable "Windows directory space" under `/usr/mywine`, then in order to be able to access this sub directory tree from Wine, you should have a drive mapping entry that maps a certain drive letter (for example, let's take drive letter `P:`) either to `/usr/mywine` or `/usr` (to also access any directories belonging to the parent directory) or `/` (to also access any directory whatsoever on this system by this drive letter mapping). The DOS drive/directory location to access files in `/usr/mywine` *in Wine* in these configuration cases would then be `P:\` or `P:\mywine` or `P:\usr\mywine`, respectively.

Installing Wine Without 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 programs without Windows installed. If you want to try it, follow these steps:

1. Make a symbolic link in `~/.wine/dosdevices` to the directory where you want `C:` to be. Refer to the wine 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.
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. Run and/or install your programs.

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 program with *`WINEDEBUG=+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.DLL/SHELL32.DLL, COMMDLG.DLL/COMDLG32.DLL and COMMCTRL.DLL/COMCTL32.DLL 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). As of now, it might not be necessary any more to use regedit.exe, since Wine has its own regedit Winelib application now.
- Also add `winhelp.exe` and `winhlp32.exe` if you want to be able to browse through your programs' help function (or in case Wine's winhelp implementation in programs/winhelp/ is not good enough, for example).

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

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 programs and their data are accessed through Wine for dual boot (Linux + Windows) systems.

Wine maps mounted FAT file systems, such as `/c`, to drive letters, such as “c:”, by means of symbolic links in the `dosdevices` directory. Thus, in your dosdevices directory, you could type the command:

```
ln -s /c c:
```

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 programs 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 programs that Wine runs unrestricted access to FAT files systems. Running wine as root also allows programs 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 program can do to the system, so it’s *HIGHLY DISCOURAGED*.

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 user group (`gid`).

Shadowing FAT filesystems

Shadowing provides a finer granularity of control. Parts of the original FAT filesystem can be copied so that the program can safely work with those copied parts while the program continues to directly read the remaining parts. This is done with symbolic links. For example, consider a system where a program 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' ~/.wine/config
```

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.

Drive labels and serial numbers

Wine can read drive volume labels and serial numbers directly from the device. This may be useful for many Win 9x games or for setup programs distributed on CD-ROMs that check for volume label.

What's Supported?

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

How To Set Up?

Reading labels and serial numbers just works automatically if you specify the correct symbolic links for the devices (with double colons after the drive letters) in your `dosdevices` directory. Note that the device has to exist and must be accessible by the user running Wine if you do this, though.

If you don't want to read labels and serial numbers directly from the device, you can create files at the root of the drive named `.windows-label` and `.windows-serial` respectively. These are simple ASCII files that you can create with any text editor; the label can be set to any string you like, the serial number should be expressed as an hexadecimal number.

Examples

Here's a simple example of CD-ROM and floppy:

```
cd ~/.wine/dosdevices

ln -s /mnt/floppy a:
ln -s /dev/fd0 a::

ln -s /mnt/cdrom r:
ln -s /dev/hda1 r::
```

Todo / Open Issues

- The CD-ROM label can be read only if the data track of the disk resides in the first track and the cdrom is iso9660.
- Support for labels/serial nums WRITING.
- What about reading ext2 volume label? ....

The 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/XP/etc.-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.

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 `wine.inf`. 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 **regedit** tool to be found in the `programs/regedit/` directory in Wine source. `wine.inf` is applied even if you plan to use a native Windows registry, since Wine needs some specific registry settings in its registry (for special workarounds for certain programs etc.). This is done automatically by wine the first time you run it.

Using a Windows registry

If you point Wine at an existing 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. In the Wine config file, there are a number of configuration settings in the [registry] section (see below) specific to the handling of Windows registry content by Wine.

The Registry

The initial default registry content to be used by the Wine registry files is in the file `wine.inf`. It contains directory paths, class IDs, and more; it must be installed before most `INSTALL.EXE` or `SETUP.EXE` applications will work.

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.

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 configuration 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 `user.reg` 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 user.reg /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 user.reg
```

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 [registry] section

Now let's look at the Wine configuration file 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.

DLL configuration

Introduction

If your programs don't work as expected, then it's often because one DLL or another is failing. This can often be resolved by changing certain DLLs from Wine built-in to native Windows DLL file and vice versa.

A very useful help to find out which DLLs are loaded as built-in and which are loaded as native Windows file can be the debug channel loaddll, activated via the environment variable **WINEDEBUG=+loaddll**.

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 too 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. Has became obsolete, ignored.

elfdll

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

DLL Overrides

The wine configuration 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 "built-in" 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.

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 built-ins 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):

krnl386, kernel32

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

gdi, gdi32

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

user, user32

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

ntdll

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

w32skrnl

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

wow32

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

system

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

display

Display driver. Definitely leave at `builtin`.

toolhelp

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

ver, version

Versioning. Seldom useful to mess with.

advapi32

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

commdlg, comdlg32

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

commctrl, comctl32

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

shell, shell32

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

winsock, wsock32

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

icmp

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

mpr

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

lzexpand, lz32

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

winaspi, wnaspi32

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

crtdll

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

winspool.drv

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

ddraw

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

dinput

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

dsound

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

dplay/dplayx

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

mmsystem, winmm

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

msacm, msacm32

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

msvideo, msvfw32

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

mcicda.drv

CD Audio MCI driver.

mciseq.drv

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

mciwave.drv

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

mciavi.drv

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

mcianim.drv

Animation MCI driver.

msacm.drv

Audio Compression Manager. Set to same as msacm32.

midimap.drv

MIDI Mapper.

wprocs

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

System DLLs

The Wine team has determined that it is necessary to create fake DLL files to trick many programs 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 program think it's there, and Wine's built-in DLL will be loaded when the program 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).

Missing DLLs

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 program 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 configuration 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

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.

Configuring the graphics driver (x11drv, ttydrv etc.)

Wine currently supports several different display subsystems (graphics / text) that are available on various operating systems today. For each of these, Wine implements its own interfacing driver. This section explains how to select one of these drivers and how to further configure the respective driver. Once you're finished with that, you can consider your Wine installation to be finished.

The display drivers currently implemented in Wine are: x11drv, which is used for interfacing to X11 graphics (the one you'll most likely want to use) and ttydrv (used for text mode console apps mainly that don't really need any graphics output). Once you have decided which display driver to use, it is chosen with the `GraphicsDriver` option in the [wine] section of `~/.wine/config`.

Configuring the x11drv graphics driver

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 programs 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 configuration 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 programs 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 programs 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 configuration 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 program windows will just be Wine-drawn windows inside the Wine-provided desktop window (which will itself be managed by your window manager), and Windows programs 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 program; 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 programs 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`.

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 programs.

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.

Configuring the ttydrv graphics driver

Currently, the ttydrv doesn't have any special configuration options to set in the configuration file.

Setting the Windows and DOS version value

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 configuration 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).

Dealing with Fonts

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.org/development/

How To Convert Windows Fonts

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

1. Extract bitmap fonts with **fnt2bdf**.
2. Convert `.bdf` files produced by Step 1 into `.pcf` files with **bdftopcf**.
3. Copy `.pcf` files to the font server directory which is usually `/usr/lib/X11/fonts/misc` (you will probably need superuser privileges). If you want to create a new font directory you will need to add it to the font path.
4. Run **mkfontdir** for the directory you copied fonts to. If you are already in X you should run **xset fp rehash** to make X server aware of the new fonts. You may also or instead have to restart the font server (using e.g. **/etc/init.d/xfs restart** under Red Hat 7.1)
5. Edit the `~/.wine/config` file to remove aliases for the fonts you've just installed.

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

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

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

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

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

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

```
AliasN = [Windows font], [X font] <, optional "mask X font" flag>
```

Example:

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

Comments:

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

X font	...will show up as...	Extracted name
--international-...	->	"International"
-adobe-helvetica-...	->	"Helvetica"
-adobe-utopia-...	->	"Utopia"
-misc-fixed-...	->	"Fixed"
-...	->	
-sony-fixed-...	->	"Sony Fixed"
-...	->	

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

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

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

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

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

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

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

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

Example:

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

Comments:

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

How To Manage Cached Font Metrics

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

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

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

Too Small Or Too Large Fonts

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

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

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

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

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

Setting up a TrueType Font Server

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

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

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

In `/c/windows/fonts/fonts.scale` I have:

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

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

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

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

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

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

Printing in Wine

How to print documents in Wine...

Printing

Printing in Wine can be done using the built-in Wine PostScript driver (+ ghostscript to produce output for non-PostScript printers).

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

Built-in 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.

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

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 customize the PostScript driver contents of `wine.inf` (see below) and use the Winelib program **programs/regedit/regedit**. For example, if you have installed the Wine source tree in `/usr/src/wine`, you could use the following series of commands:

- **`#vi /usr/share/wine/wine.inf`**
- Edit the copy of `wine.inf` to suit your PostScript printing requirements. At a minimum, you must specify a PPD file for each printer.
- **`$wineprefixcreate`**

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 built-in.

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 built-in 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.

SCSI Support

This file describes setting up the Windows ASPI interface. 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!

Windows requirements

1. The software needs to use the "Adaptec" compatible drivers (ASPI). At least with Mustek, they allow you the choice of using the built-in 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.
2. You probably need a real windows install of the software to set the LUN's/SCSI id's up correctly. I'm not exactly sure.

Linux requirements

1. Your SCSI card must be supported under Linux. This will not work with an unknown SCSI card. Even for cheap'n crappy "scanner only" controllers some special Linux drivers exist on the net. If you intend to use your IDE device, you need to use the ide-scsi emulation. Read http://www.linuxdoc.org/HOWTO/CD-Writing-HOWTO.html (http://www.linuxdoc.org/HOWTO/CD-Writing-HOWTO.html) for ide-scsi setup instructions.
2. Compile generic SCSI drivers into your kernel.
3. This seems to be not required any more for newer (2.2.x) kernels: Linux by default uses smaller SCSI buffers than Windows. There is a kernel build define `SG_BIG_BUFF` (in `sg.h`) that is by default set too low. The SANE project recommends `130560` and this seems to work just fine. This does require a kernel rebuild.
4. Make the devices for the scanner (generic SCSI devices) - look at the SCSI programming HOWTO at http://www.linuxdoc.org/HOWTO/SCSI-Programming-HOWTO.html (http://www.linuxdoc.org/HOWTO/SCSI-Programming-HOWTO.html) for device numbering.
5. I would recommend making the scanner device writable by a group. I made a group called `scanner` and added myself to it. Running as root increases your risk of sending bad SCSI commands to the wrong device. With a regular user, you are better protected.
6. For Win32 software (WNASPI32), Wine has auto-detection in place. For Win16 software (WINASPI), you need to add a SCSI device entry for your particular scanner to ~/.wine/config. The format is `[scsi cCtTdD]` where `"C" = "controller"`, `"T" = "target"`, `D=LUN`

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

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

   Yours will vary with your particular SCSI setup.

Notes

The biggest drawback 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]

Using ODBC

This section describes how ODBC works within Wine and how to configure it.

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 built-in 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 built-in odbc32 DLL. The built-in odbc32 (currently) looks to the environment 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 environment 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 built-in version of odbc32.dll, by modifying the DLL configuration. This built-in DLL merely acts as a stub between the calling code and the Unix ODBC library.

If you have any problems then you can use WINEDEBUG=+odbc32 command before running wine 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

Native ODBC drivers have been reported to work for many types of databases including MSSQL and Oracle. In fact, some like MSSQL can only be accessed on Linux through a Winelib app. Rather than just copying DLL files, most ODBC drivers require a Windows-based installer to run to properly configure things such as registry keys.

In order to set up MSSQL support you will first need to download and run the mdac_typ.exe installer from microsoft.com. In order to configure your ODBC connections you must then run CLICONFG.EXE and ODBCAD32.EXE under Wine. You can find them in the windows\system directory after mdac_typ runs. Compare the output of these programs with the output on a native Windows machine. Some things, such as protocols, may be missing because they rely on being installed along with the operating system. If so, you may be able to copy missing functionality from an existing Windows installation as well as any registry values required. A native Windows installation configured to be used by Wine should work the same way it did when run natively.

Types successfully tested under wine:

DB Type	Usefulness
MS SQL	100%

Please report any other successes to the wine-devel (mailto:wine-devel@winehq.org) mailing list.

Chapter 6. Running Wine

This chapter will describe all aspects of running Wine, like e.g. basic Wine invocation, command line parameters of various Wine support programs etc.

Basic usage: applications and control panel applets

Assuming you are using a fake Windows installation, you install applications into Wine in the same way you would in Windows: by running the installer. You can just accept the defaults for where to install, most installers will default to "C:\Program Files", which is fine. If the application installer requests it, you may find that Wine creates icons on your desktop and in your app menu. If that happens, you can start the app by clicking on them.

The standard way to uninstall things is for the application to provide an uninstaller, usually registered with the "Add/Remove Programs" control panel applet. To access the Wine equivalent, run the **uninstaller** program (it is located in the `programs/uninstaller/` directory in a Wine source directory) in a *terminal*:

```
$ uninstaller
```

Some programs install associated control panel applets, examples of this would be Internet Explorer and QuickTime. You can access the Wine control panel by running in a *terminal*:

```
$ wine control
```

which will open a window with the installed control panel applets in it, as in Windows.

If the application doesn't install menu or desktop items, you'll need to run the app from the command line. Remembering where you installed to, something like:

```
$ wine "c:\program files\appname\appname.exe"
```

will probably do the trick. The path isn't case sensitive, but remember to include the double quotes. Some programs don't always use obvious naming for their directories and EXE files, so you might have to look inside the program files directory to see what was put where.

How to run Wine

You can simply invoke the **wine** command to get a small help message:

```
Wine 20040405
Usage: wine PROGRAM [ARGUMENTS...]   Run the specified program
       wine --help                   Display this help and exit
       wine --version                Output version information and exit
```

The first argument should be the name of the file you want **wine** to execute. If the executable is in the *`Path`* environment variable, 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:

```
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 "\" !)

For details on running text mode (CUI) executables, read the section below.

Explorer-like graphical Wine environments

If you prefer using a graphical interface to manage your files you might want to consider using Winefile. This Winelib application comes with Wine and can be found with the other Wine programs. It is a useful way to view your drive configuration and locate files, plus you can execute programs directly from Winefile. Please note, many functions are not yet implemented.

Wine Command Line Options

--help

Shows a small command line help page.

--version

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

Environment variables

WINEDEBUG=[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 *`WINEDEBUG`* 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 *`WINEDEBUG`* option, separate them with commas. For example, to request *`warn`* class messages in the *`heap`* debug channel, you could invoke **wine** like this:

```
$ WINEDEBUG=warn+heap wine program_name
```

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

```
$ WINEDEBUG=heap wine program_name
```

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

```
$ WINEDEBUG=+all,-relay wine program_name
```

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

Table 6-1. Debug Channels

accel	adpcm	advapi	animate	aspi
atom	avicap	avifile	bidi	bitblt
bitmap	cabinet	capi	caret	cdrom
cfgmgr32	class	clipboard	clipping	combo
comboex	comm	commctrl	commdlg	computername
console	crtdll	crypt	curses	cursor
d3d	d3d_shader	d3d_surface	datetime	dc
ddeml	ddraw	ddraw_fps	ddraw_geom	ddraw_tex
debugstr	devenum	dialog	dinput	dll
dma	dmband	dmcompos	dmfile	dmfiledat
dmime	dmloader	dmscript	dmstyle	dmsynth
dmusic	dosfs	dosmem	dplay	dplayx
dpnhpast	driver	dsound	dsound3d	edit
enhmetafile	environ	event	eventlog	exec
file	fixup	font	fps	g711
gdi	global	glu	graphics	header
heap	hook	hotkey	icmp	icon
imagehlp	imagelist	imm	int	int21
int31	io	ipaddress	iphlpapi	jack
joystick	key	keyboard	listbox	listview
loaddll	local	mapi	mci	mcianim
mciavi	mcicda	mcimidi	mciwave	mdi
menu	menubuilder	message	metafile	midi
mmaux	mmio	mmsys	mmtime	module
monthcal	mpeg3	mpr	msacm	msdmo
msg	mshtml	msi	msimg32	msisys
msrle32	msvcrt	msvideo	mswsock	nativefont
netapi32	netbios	nls	nonclient	ntdll
odbc	ole	oledlg	olerelay	opengl
pager	palette	pidl	powermgnt	print
process	profile	progress	propsheet	psapi
psdrv	qcap	quartz	ras	rebar
reg	region	relay	resource	richedit
rundll32	sblaster	scroll	seh	selector

server	setupapi	shdocvw	shell	shlctrl
snmpapi	snoop	sound	static	statusbar
storage	stress	string	syscolor	system
tab	tape	tapi	task	text
thread	thunk	tid	timer	toolbar
toolhelp	tooltips	trackbar	treeview	ttydrv
twain	typelib	uninstaller	updown	urlmon
uxtheme	ver	virtual	vxd	wave
wc_font	win	win32	wineboot	winecfg
wineconsole	wine_d3d	winevdm	wing	winhelp
wininet	winmm	winsock	winspool	wintab
wintab32	wnet	x11drv	x11settings	xdnd
xrandr	xrender	xvidmode		

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

wineserver Command Line Options

wineserver usually gets started automatically by Wine whenever the first wine process gets started. However, wineserver has some useful command line options that you can add if you start it up manually, e.g. via a user login script or so.

-d<n>

Sets the debug level for debug output in the terminal that wineserver got started in at level <n>. In other words: everything greater than 0 will enable wineserver specific debugging output.

-h

Display wineserver command line options help message.

-k[n]

Kill the current wineserver, optionally with signal n.

-p[n]

This parameter makes wineserver persistent, optionally for n seconds. It will prevent wineserver from shutting down immediately.

Usually, wineserver quits almost immediately after the last wine process using this wineserver terminated. However, since wineserver loads a lot of things on startup (such as the whole Windows registry data), its startup might be so slow that it's very useful to keep it from exiting after the end of all Wine sessions, by making it persistent.

-w

This parameter makes a newly started wineserver wait until the currently active wineserver instance terminates.

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 are some exceptions to the rule: If you want to change the PATH, SYSTEM or TEMP variables, the of course you can't modify it that way, since this will alter the Unix environment settings. Instead, you should set them into the registry. To set them you should launch **`wine regedit`** and then go to the

```
HKEY_CURRENT_USER/Environment
```

key. Now you can create or modify the values of the variables you need

```
"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 /usr/local/wine_c_windows as Windows path, the system directory would be `/usr/local/wine_c/windows/system`. It must be set with no trailing slash, and you must be sure that you have write access to it.

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

This should be the directory you want your temp files stored in, /usr/local/wine_c/temp in our previous 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"`).

Text mode programs (CUI: Console User Interface)

Text mode programs are program which output is only made out of text (surprise!). In Windows terminology, they are called CUI (Console User Interface) executables, by opposition to GUI (Graphical User Interface) executables. Win32 API provide a complete set of APIs to handle this situation, which goes from basic features like text printing, up to high level functionalities (like full screen editing, color support, cursor motion, mouse support), going through features like line editing or raw/cooked input stream support

Given the wide scope of features above, and the current usage in Un*x world, Wine comes out with three different ways for running a console program (aka a CUI executable):

- bare streams
- wineconsole with user backend
- wineconsole with curses backend

The names here are a bit obscure. "bare streams" means that no extra support of wine is provide to map between the unix console access and Windows console access. The two other ways require the use of a specific Wine program (wineconsole) which provide extended facilities. The following table describes what you can do (and cannot do) with those three ways.

Table 6-2. Basic differences in consoles

Function	Bare streams	Wineconsole & user backend	Wineconsole & curses backend
How to run (assuming executable is called foo.exe)	`$ wine foo.exe`	`$ wineconsole -- --backend=user foo.exe`	`$ wineconsole foo.exe` You can also use --backend=curses as an option
Good support for line oriented CUI applications (which print information line after line)	Yes	Yes	Yes
Good support for full screen CUI applications (including but not limited to color support, mouse support...)	No	Yes	Yes
Can be run even if X11 is not running	Yes	No	Yes
Implementation	Maps the standard Windows streams to the standard Unix streams (stdin/stdout/stderr)	Wineconsole will create a new Window (hence requiring the USER32 DLL is available) where all information will be displayed	Wineconsole will use existing unix console (from which the program is run) and with the help of the (n)curses library take control of all the terminal surface for interacting with the user
Known limitations			Will produce strange behavior if two (or more) Windows consoles are used on the same Un*x terminal.

Configuration of CUI executables

When wineconsole is used, several configuration options are available. Wine (as Windows do) stores, on a per application basis, several options in the registry. This let a user, for example, define the default screen-buffer size he would like to have for a given application.

As of today, only the USER backend allows you to edit those options (we don't recommend editing by hand the registry contents). This edition is fired when a user right click in the console (this popups a menu), where you can either choose from:

- Default: this will edit the settings shared by all applications which haven't been configured yet. So, when an application is first run (on your machine, under your account) in wineconsole, wineconsole will inherit this default settings for the application. Afterwards, the application will have its own settings, that you'll be able to modify at your will.

Properties: this will edit the application's settings. When you're done, with the edition, you'll be prompted whether you want to:

1. Keep these modified settings only for this session (next time you run the application, you will not see the modification you've just made).
2. Use the settings for this session and save them as well, so that next you run your application, you'll use these new settings again.

Here's the list of the items you can configure, and their meanings:

Table 6-3. Wineconsole configuration options

Configuration option	Meaning
Cursor's size	Defines the size of the cursor. Three options are available: small (33% of character height), medium (66%) and large (100%)
Popup menu	It's been said earlier that wineconsole configuration popup was triggered using a right click in the console's window. However, this can be an issue when the application you run inside wineconsole expects the right click events to be sent to it. By ticking control or shift you select additional modifiers on the right click for opening the popup. For example, ticking shift will send events to the application when you right click the window without shift being hold down, and open the window when you right-click while shift being hold down.
Quick edit	This tick box lets you decide whether left-click mouse events shall be interpreted as events to be sent to the underlying application (tick off) or as a selection of rectangular part of the screen to be later on copied onto the clipboard (tick on).
History	This lets you pick up how many commands you want the console to recall. You can also drive whether you want, when entering several times the same command - potentially intertwined with others - whether you want to store all of them (tick off) or only the last one (tick on).
Police	The Police property sheet allows you to pick the default font for the console (font file, size, background and foreground color).

Configuration option	Meaning
Screenbuffer & window size	The console as you see it is made of two different parts. On one hand there's the screenbuffer which contains all the information your application puts on the screen, and the window which displays a given area of this screen buffer. Note that the window is always smaller or of the same size than the screen buffer. Having a strictly smaller window size will put on scrollbars on the window so that you can see the whole screenbuffer's content.
Close on exit	If it's ticked, then the wineconsole will exit when the application within terminates. Otherwise, it'll remain opened until the user manually closes it: this allows seeing the latest information of a program after it has terminated.
Edition mode	When the user enter commands, he or she can choose between several edition modes: • Emacs: the same keybindings as under emacs are available. For example, Ctrl-A will bring the cursor to the beginning of the edition line. See your emacs manual for the details of the commands. • Win32: these are the standard Windows console key-bindings (mainly using arrows).

Chapter 7. 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 ?)

Verify your wine configuration

Refer to the Configuration verification section

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 WINEDEBUG=+loaddll to figure out which DLLs are being used, and whether they're being loaded as native or built-in. 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

There is a really good chance that someone has already tried to do the same thing as you. You may find the following resources helpful:

- Search WineHQ's Application Database (http://appdb.winehq.org) to check for any tips relating to the program. If your specific version of the program isn't listed you may find a different one contains enough information to help you out.
- Frank's Corner (http://www.frankscorner.org) contains a list of applications and detailed instructions for setting them up. Further help can be found in the user forums.
- Google (http://www.google.com) can be useful depending on how you use it. You may find it helpful to search Google Groups (http://groups.google.com), in particular the comp.emulators.ms-windows.wine (http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&group=comp.emulators.ms-windows.wine) group.
- Freenode.net (http://www.freenode.net) hosts an IRC channel for Wine. You can access it by using any IRC client such as Xchat. The settings you'll need are: server = irc.freenode.net, port = 6667, and channel = #winehq
- If you have a program that needs the Visual Basic Runtime Environment, you can download it from this Microsoft site (http://www.microsoft.com/downloads/details.aspx?FamilyID=bf9a24f9-b5c5-48f4-8edd-cdf2d29a79d5&DisplayLang=en/)
- If you know you are missing a DLL, such as mfc42, you may be able to find it at www.dll-files.com (http://www.dll-files.com/)
- Wine's mailing lists (http://www.winehq.org/site/forums#ml) may also help, especially wine-users. The wine-devel list may be appropriate depending on the type of problem you are experiencing. If you post to wine-devel you should be prepared to do a little work to help diagnose the problem. Read the section below to find out how to debug the source of your problem.
- If all else fails, you may wish to investigate commercial versions of Wine to see if your application is supported.

Debug it!

Finding the source of your problem is the next step to take. There is a wide spectrum of possible problems ranging from simple configurations issues to completely unimplemented functionality in Wine. The next section will describe how to file a bug report and how to begin debugging a crash. For more information on using Wine's debugging facilities be sure 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.org/). 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 Red Hat 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 WINEDEBUG environment variable *`WINEDEBUG=+relay`* option (i.e., **WINEDEBUG=+relay wine 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.org/). 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 *WINEDEBUG* 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 | WINEDEBUG=+relay wine [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:

```
$ WINEDEBUG=+relay wine [other_options] program_name |& tee filename.out;
$ tail -n 100 filename.out > report_file
```

bash and other sh-like shells:

```
$ WINEDEBUG=+relay wine [other_options] program_name 2>&1 | tee filename.out;
$ tail -n 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.org/). 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.

Glossary

Binary

A file which is in machine executable, compiled form: hex data (as opposed to a source code file).

CVS

Concurrent Versions System, a software package to manage software development done by several people. See the CVS chapter in the Wine Developers Guide for detailed usage information.

Distribution

A distribution is usually the way in which some "vendor" ships operating system CDs (usually mentioned in the context of Linux). A Linux environment can be shipped in lots of different configurations: e.g. distributions could be built to be suitable for games, scientific applications, server operation, desktop systems, etc.

DLL

A DLL (Dynamic Link Library) is a file that can be loaded and executed by programs dynamically. Basically it's an external code repository for programs. Since usually several different programs reuse the same DLL instead of having that code in their own file, this dramatically reduces required storage space. A synonym for a DLL would be library.

Editor

An editor is usually a program to create or modify text files. There are various graphical and text mode editors available on Linux.

Examples of graphical editors are: nedit, gedit, kedit, xemacs, gxedit.

Examples of text mode editors are: joe, ae, emacs, vim, vi. In a *terminal*, simply run them via:

```
$ editorname
filename
```

Environment variable

Environment variables are text definitions used in a *Shell* to store important system settings. In a **bash** shell (the most commonly used one in Linux), you can view all environment variables by executing:

```
set
```

If you want to change an environment variable, you could run:

```
export MYVARIABLE=mycontent
```

For deleting an environment variable, use:

```
unset MYVARIABLE
```

Package

A package is a compressed file in a *distribution* specific format. It contains the files for a particular program you want to install. Packages are usually installed via the **dpkg** or **rpm** package managers.

root

root is the account name of the system administrator. In order to run programs as root, simply open a *Terminal* window, then run:

```
$ su -
```

This will prompt you for the password of the root user of your system, and after that you will be able to system administration tasks that require special root privileges. The root account is indicated by the

```
#
```

prompt, whereas '$' indicates a normal user account.

Shell

A shell is a tool to enable users to interact with the system. Usually shells are text based and command line oriented. Examples of popular shells include **bash**, **tcsh** and **ksh**. Wine assumes that for Wine installation tasks, you use **bash**, since this is the most popular shell on Linux. Shells are usually run in a *Terminal* window.

Source code

Source code is the code that a program consists of before the program is being compiled, i.e. it's the original building instructions of a program that tell a compiler what the program should look like once it's been compiled to a *Binary*.

Terminal

A terminal window is usually a graphical window that one uses to execute a **Shell**. If Wine asks you to open a terminal, then you usually need to click on an icon on your desktop that shows a big black window (or, in other cases, an icon displaying a maritime shell). Wine assumes you're using the **bash** shell in a terminal window, so if your terminal happens to use a different shell program, simply type:

```
bash
```

in the terminal window.