Winelib User’s Guide

Winelib User's Guide

Table of Contents

Chapter 1. Winelib Introduction

What is Winelib?

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

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

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

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

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

System requirements

The requirements for Winelib are similar to those for Wine.

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

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

Getting Started

Winemaker introduction

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

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

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

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

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

Let's suppose that you are already in the top directory of your sources. Then converting your project to Winelib may be as simple as just running the three commands below (note the dot indicating current directory at the end of the first command):

```
$ winemaker --lower-uppercase .
$ make
```

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

Test Drive

Before starting to work on a big project you may want to try to port a small application. The notepad application from the Wine source tree suits well for testing purposes. It can be found in the programs subdirectory. notepad is a simple application, but has a few C, header and resource files.

Run **make clean** in the notepad source directory if it contains results of previous builds. Create a separate directory with name notepad2, so it won't conflict with the Wine copy of the application. Copy the sources of notepad (files *.c, *.h, *.rc) to this directory. Now run the commands, mentioned above from the notepad2 directory:

```
$ winemaker --lower-uppercase .
$ make
```

Notice how the build fails in a resource file (`Da.rc` at the time this was written). This is because for the time being, the Wine resource compiler (**windres**) can't cope with more than one resource file ending up in a given executable. To get around that limitation, the Wine developers have chosen to only have one master resource file per application, and include the other ones via `#include` statements in the master one. The included files do not `#include` the necessary files to be compilable on their own. You will need to do the same thing for now in your own Winelib port projects.

To fix that problem, you will need to edit the list of resource files winemaker thought notepad used, and only keep the master resource file. To do so, open `notepad2/Makefile` in a text editor and search for an assignment to a variable with `RC_SRCS` as part of its name. It will likely be named `notepad2_exe_RC_SRCS`. There will be a list of resource files starting on the same line. Remove them all except `rsrc.rc` (this is the master one for notepad). Save your work.

Now you're ready to continue at the same point the first **make** failed. Return to the notepad2 directory and type:

```
$ make
```

You are done! Now you can start the application as **./notepad2**.

If you come across problems preparing and building this application this probably means that winemaker utility is broken by some changes in Wine. Try asking for help on `<wine-devel@winehq.org>`

Step by step guide

Let's retrace the steps above in more details.

`Getting the source`

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

`cd <root_dir>`

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

`Making the source writable`

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

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

`Running winemaker`

Then you'll run winemaker. Here are the options you will most likely want to use. For complete list of options see the winemaker man page.

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

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

`--nobackup`

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

`--dll`
`--console`

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

`--mfc`

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

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

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

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

`File renaming`

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

`Source modifications and makefile generation`

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

`Running make`

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

- review the `Makefile` files to adjust what winemaker thinks are the binaries you are trying to build and which sources should be used for each. See the *Winemaker's source analysis* section for some hints.
- fix the portability issues in your sources. See *Portability issues* for more details.

Chapter 2. Portability issues

Unicode

The `wchar_t` type has different standard sizes in Unix (4 bytes) and Windows (2 bytes). You need a recent gcc version (2.9.7 or later) that supports the *-fshort-wchar* option to set the size of `wchar_t` to the one expected by Windows applications.

If you are using Unicode and you want to be able to use standard library calls (e.g. `wcslen`, `wsprintf`), then you must use the msvcrt runtime library instead of glibc. The functions in glibc will not work correctly with 16 bit strings.

C library

There are 2 choices available to you regarding which C library to use: the native glibc C library or the msvcrt C library.

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

Using glibc in general has the lowest overhead, but this is really only important for file I/O, as many of the functions in msvcrt are simply resolved to glibc.

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

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

In these cases you should use msvcrt to provide your C runtime calls.

```
import msvcrt.dll
```

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

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

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

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

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

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

Compiling Problems

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

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

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

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

Chapter 3. The Winelib development toolkit

Winemaker

Support for Visual C++ projects

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

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

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

Winemaker's source analysis

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The interactive mode

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

The Makefile files

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

```
### Generic autoconf variables

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

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

```
SUBDIRS               =
DLLS                  =
EXES                  = hello.exe
```

This is where the targets for this directory are listed. The names are pretty self-explanatory. `SUBDIRS` is usually only present in the top-level makefile. For libraries and executables, specify the full name, including the '.dll' or '.exe' extension. Note that these names must be in all lowercase.

```
### Global settings

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

This section contains the global compilation settings: they apply to all the targets in this makefile. The `LIBRARIES` variable allows you to specify additional Unix libraries to link with. Note that you would normally not specify Winelib libraries there. To link with a Winelib library, one uses the `DLLS` variables of the Makefile. The exception is for C++ libraries where you currently don't have a choice but to link with them in the Unix sense. One library you are likely to find here is `mfc` (note, the '-l' is omitted).

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

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

```
### hello.exe sources and settings

hello_exe_C_SRCS          = hello.c
hello_exe_CXX_SRCS        =
hello_exe_RC_SRCS         =
hello_exe_SPEC_SRCS       =
```

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

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

```
hello_exe_DLL_PATH        =
hello_exe_DLLS            =
```

```
hello_exe_LIBRARY_PATH    =
hello_exe_LIBRARIES       =
hello_exe_DEPENDS         =
```

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

The `DLLS` field is where you would enumerate the list of DLLs that executable imports. It should contain the full DLL name including the '.dll' extension, but not the '-l' option.

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

```
hello_exe_OBJS        = $(hello_exe_C_SRCS:.c=.o) \
                        $(hello_exe_CXX_SRCS:.cpp=.o) \
                        $(EXTRA_OBJS)
```

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

```
### Global source lists

C_SRCS                = $(hello_exe_C_SRCS)
CXX_SRCS              = $(hello_exe_CXX_SRCS)
RC_SRCS               = $(hello_exe_RC_SRCS)
SPEC_SRCS             = $(hello_exe_SPEC_SRCS)
```

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

> **Note:** FIXME:The following is not up-to-date.

```
### Generic autoconf targets

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

@MAKE_RULES@

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

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

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

```
### Target specific build rules

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

$(hello_SPEC_SRCS:.spec=.spec.c): $(hello_SPEC_SRCS:.spec) $(hello_SPEC_SRCS:.spec=.tmp.o) $(hello_RC_SRCS:.rc=.res)
```

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

hello.so: $(hello_SPEC_SRCS:.spec=.spec.o) $(hello_OBJS) $(hello_DEP
ENDS)
        $(LDSHARED) $(LDDLLFLAGS) -o $@ $(hello_OBJS) $(hello_SPEC_SRCS:.spec=.spec.o) $(hello_LIBRARY_PATH) $(hello_LIBRARIES:%=-l%) $(DLL_LINK) $(LIBS)
        test -f hello || $(LN_S) $(WINE) hello
```

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

Compiling resource files: WRC

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

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

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

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

.SUFFIXES: .rc .res

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

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

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

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

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

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

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

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

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

Compiling message files: WMC

how does one use it???

The Spec file

Introduction

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

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

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

A typical spec file will look something like this:

```
init    WinMain
rsrc    resource.res
```

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

init

`init` defines what is the entry point of that module. For a library this is customarily set to `DllMain`, for a console application this is `main` and for a graphical application this is `WinMain`.

rsrc

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

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

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

@

Note: FIXME: You must now export functions from DLLs.

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

Compiling it

Note: FIXME: This section is very outdated and does not correctly describe the current use of winebuild and spec files. In particular, with recent versions of winebuild most of the information that used to be in the spec files is now specified on the command line.

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

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

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

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

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

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

More details

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

```
# comment text
```

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

```
init    FUNCTION
```

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

```
rsrc    RES_FILE
```

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

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

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

```
ORDINAL equate EXPORTNAME DATA
```

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

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

This field can be present zero or more times. Each instance defines a function entry point. The prototype defined by `EXPORTNAME ([ARGTYPE [ARGTYPE [...]]])` specifies the name available for dynamic linking and the format of the arguments. `ORDINAL` is replaced by the ordinal number corresponding to the function, or `@` for automatic ordinal allocation (Win32 only).

`FUNCTYPE` should be one of:

pascal16

for a Win16 function returning a 16-bit value

pascal

for a Win16 function returning a 32-bit value

register

for a function using CPU register to pass arguments

interrupt

for a Win16 interrupt handler routine

stdcall

for a normal Win32 function

cdecl

for a Win32 function using the C calling convention

varargs

for a Win32 function taking a variable number of arguments

`ARGTYPE` should be one of:

word

for a 16 bit word

long

a 32 bit value

ptr

for a linear pointer

str

for a linear pointer to a null-terminated string

s_word

for a 16 bit signed word

segptr

for a segmented pointer

segstr

for a segmented pointer to a null-terminated string

wstr

for a linear pointer to a null-terminated wide (16-bit Unicode) string

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

Strings should almost always map to str, wide strings - wstr. As the general rule it depends on whether the parameter is IN, OUT or IN/OUT.

- IN: str/wstr
- OUT: ptr
- IN/OUT: str/wstr

It is for debug messages. If the parameter is OUT it might not be initialized and thus it should not be printed as a string.

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

To declare a function using a variable number of arguments in Win16, specify the function as taking no arguments. The arguments are then available with CURRENT_STACK16->args. In Win32, specify the function as `varargs` and declare it with a '...' parameter in the C file. See the wsprintf* functions in `user.spec` and `user32.spec` for an example.

```
ORDINAL stub EXPORTNAME
```

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

```
ORDINAL extern EXPORTNAME SYMBOLNAME
```

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

```
ORDINAL forward EXPORTNAME SYMBOLNAME
```

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

Linking it all together

> **Note:** FIXME:The following is not up-to-date.

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

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

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

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

Chapter 4. Dealing with the MFC

Introduction

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

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

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

Legal issues

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

During the compilation of your program, you will be combining code from several sources: your code, Winelib code, Microsoft MFC code, and possibly code from other vendor sources. As a result, you must ensure that the licenses of all code sources are obeyed. What you are allowed and not allowed to do can vary depending on how you combine the code and if you will be distributing it. For example, if you are releasing your code under the GPL or LGPL, you cannot use MFC because these licenses do not allow covered code to depend on libraries with non-compatible licenses. There is a workaround - in the license for your code you can make an exception for the MFC library. For details see The GNU GPL FAQ (http://www.gnu.org/licenses/gpl-faq.html).

Wine/Winelib is distributed under the GNU Lesser General Public License. See the license for restrictions on the modification and distribution of Wine/Winelib code. In general it is possible to satisfy these restrictions in any type of application. On the other hand, MFC is distributed under a very restrictive license and the restrictions vary from version to version and between service packs. There are basically three aspects you must be aware of when using the MFC.

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

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

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

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

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

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

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

Compiling the MFC

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

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

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

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

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

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

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

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

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

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

Chapter 5. Building WineLib DLLs

Introduction

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

- You are porting a large application that uses several third-party libraries. One is available on Linux but you are not yet ready to link to it directly as a Linux shared library.
- There is a well-defined interface available and there are several Linux solutions that are available for it (e.g. the ODBC interface in Wine).
- You have a binary only Windows application that can be extended through plugins, such as a text editor or IDE.

The process for dealing with these situations is actually quite simple. You need to write a spec file that will describe the library's interface in the same format as a Dll (primarily what functions it exports). Also you will want to write a small wrapper around the library. You combine these to form a Wine built-in Dll that links to the Linux library. Then you modify the DllOverrides in the wine config file to ensure that this new built-in DLL is called rather than any windows version.

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

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

Specifically, the application calls a function:

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

and the linux library exports a corresponding function:

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

Writing the spec file

Start by writing the spec file. This file will describe the interface as if it were a DLL. See elsewhere for the details of the format of a spec file (e.g. man winebuild).

In the simple example we want a Wine built-in Dll that corresponds to the MyWin Dll. The spec file is `MyWin.dll.spec` and looks something like this:

```
#
# File: MyWin.dll.spec
#
# some sort of copyright
#
```

```
# Wine spec file for the MyWin.dll built-in library (a minimal wrapper around the
# linux library libMyLinux)
#
# For further details of wine spec files see the Winelib documentation at
# www.winehq.org

2 stdcall MyWinFunc (long ptr ptr ptr ptr long long ptr) MyProxyWinFunc

# End of file
```

Notice that the arguments are flagged as long even though they are smaller than that. With this example we will link directly to the Linux shared library whereas with the ODBC example we will load the Linux shared library dynamically.

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

Writing the wrapper

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

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

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

/* This declaration is as defined in the spec file.  It is deliberately not
 * specified in terms of <3rd party> types since we are messing about here
 * between two operating systems (making it look like a Windows thing when
 * actually it is a Linux thing).  In this way the compiler will point out any
 * inconsistencies.
 * For example the fourth argument needs care
 */
signed short WINAPI MyProxyWinFunc (unsigned short a, void *b, void *c,
                unsigned long *d, void *e, int f, char g, unsigned char *h)
{
    unsigned short d1;
```

```
    signed short ret;

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

    return ret;
}

/* End of file */
```

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

`DllMain` the function is used to initialize the DLL. On the process attach event the function dynamically links to the desired Linux ODBC library (since there are several available) and builds a list of function pointers. It unlinks on the process detach event.

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

Building

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

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

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

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

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

Installing

So how do you install the proxy and ensure that everything connects up correctly? You have quite a bit of flexibility in this area so what follows are not the only options available.

Ensure that the actual Linux Shared Object is placed somewhere where the Linux system will be able to find it. Typically this means it should be in one of the directories mentioned in the /etc/ld.so.conf file or somewhere in the path specified by LD_LIBRARY_PATH. If you can link to it from a Linux program it should be OK.

Put the proxy shared object (MyWin.dll.so) in the same place as the rest of the built-in DLLs. (If you used winemaker to set up your build environment then running "make install" as root should do that for you) Alternatively ensure that WINEDLLPATH includes the directory containing the proxy shared object.

If you have both a Windows DLL and a Linux DLL/proxy pair then you will have to ensure that the correct one gets called. The easiest way is probably simply to rename the windows version so that it doesn't get detected. Alternatively you could specify in the DllOverrides section (or the AppDefaults\\myprog.exe\\DllOverrides section) of the config file (in your .wine directory) that the built-in version be used. Note that if the Windows version Dll is present and is in the same directory as the executable (as opposed to being in the Windows directory) then you will currently need to specify the whole path to the dll, not merely its name.

Once you have done this you should be using the Linux Shared Object successfully. If you have problems then set the WINEDEBUG=+module environment variable before running wine to see what is actually happening.

Converting filenames

Suppose you want to convert incoming DOS format filenames to their Unix equivalent. Of course there is no suitable function in the true Microsoft Windows API, but wine provides a function for just this task and exports it from its copy of the kernel32 DLL. The function is `wine_get_unix_file_name` (defined in winbase.h).