

Wine Developer's Guide

Wine Developer's Guide

Table of Contents

I. Developing Wine.....	1
1. Debugging Wine.....	1
Introduction.....	1
WineDbg's modes of invocation.....	1
Using the Wine Debugger	3
Useful memory addresses	9
Configuration	10
WineDbg Command Reference	12
Other debuggers	16
Limitations.....	18
2. Documenting Wine	21
An Overview Of Wine Documentation.....	21
Writing Wine API Documentation	21
The Wine DocBook System	27
3. Submitting Patches.....	39
Patch Format	39
Some notes about style.....	39
Quality Assurance	40
4. Writing Conformance tests	41
Introduction.....	41
What to test for?	41
Running the tests in Wine.....	42
Cross-compiling the tests with MinGW	42
Building and running the tests on Windows.....	43
Inside a test	44
Writing good error messages	45
Handling platform issues	46
5. Internationalization.....	49
Adding New Languages.....	49
II. Wine Architecture.....	51
6. Overview	51
Basic Overview	51
Module Overview.....	53
Wine/Windows DLLs.....	58
7. Debug Logging	61
Debugging classes	61
Debugging channels.....	61
Are we debugging?	62
Helper functions	62
Controlling the debugging output	63
Compiling Out Debugging Messages.....	64
A Few Notes on Style.....	65
8. COM/OLE in Wine.....	67
Writing OLE Components for Wine.....	67
9. Wine and OpenGL	73
What is needed to have OpenGL support in Wine.....	73
How it all works	73
Known problems	75
10. Outline of DirectDraw Architecture	77
DirectDraw inheritance tree.....	77
DirectDrawSurface inheritance tree.....	77
Interface Thunks	77
Logical Object Layout	78
Creating Objects.....	78
11. Wine and Multimedia	79
Overview.....	79
Low level layers	79

Mid level drivers (MCI)	83
High level layers	85
Multimedia configuration	87
Multimedia architecture	88
MS ACM DLLs	90
III. Advanced Topics	93
12. Low-level Implementation.....	93
Keyboard.....	93
Undocumented APIs.....	94
Accelerators	96
Doing A Hardware Trace.....	96
13. Porting Wine to new Platforms	101
Porting Wine to new Platforms	101
14. Consoles in Wine	103
15. How to do regression testing using CVS	105

Chapter 1. Debugging Wine

Introduction

Processes and threads: in underlying OS and in Windows

Before going into the depths of debugging in Wine, here's a small overview of process and thread handling in Wine. It has to be clear that there are two different beasts: processes/threads from the Unix point of view and processes/threads from a Windows point of view.

Each Windows' thread is implemented as a Unix process (under Linux using the `clone` syscall), meaning that all threads of a same Windows' process share the same (unix) address space.

In the following:

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

A `W-process` is made of one or several `W-threads`. Each `W-thread` is mapped to one and only one `U-process`. All `U-processes` of a same `W-process` share the same address space.

Each Unix process can be identified by two values:

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

Each Windows' process has also a Windows' process id (`wpid` in the following). It must be clear that `upid` and `wpid` are different and shall not be used instead of the other.

`wpid` and `tid` are defined (Windows) system wide. They must not be confused with process or thread handles which, as any handle, is an indirection to a system object (in this case process or thread). A same process can have several different handles on the same kernel object. The handles can be defined as local (the values is only valid in a process), or system wide (the same handle can be used by any `W-process`).

Wine, debugging and WineDbg

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

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

Wine implements most of the Windows' debugging API (the part in `KERNEL32.DLL`, not the one in `IMAGEHLP.DLL`), and allows any program (emulated or Winelib) using that API to debug a `W-process`.

WineDbg is a Winelib application making use of this API to allow debugging both any Wine or Winelib applications as well as Wine itself (kernel and all DLLs).

WineDbg's modes of invocation

Starting a process

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

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

Attaching

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

On exceptions

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

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

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

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

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

continue:

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

pass:

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

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

Interrupting

You can stop the debugger while it's running by hitting Ctrl-C in its window. This will stop the debugged process, and let you manipulate the current context.

Quitting

Wine supports the new XP APIs, allowing for a debugger to detach from a program being debugged (see **detach** command). Unfortunately, as the debugger cannot, for now, neither clear its internal information, nor restart a new process, the debugger, after detaching itself, cannot do much except being quitted.

Using the Wine Debugger

This section describes where to start debugging Wine. If at any point you get stuck and want to ask for help, please read the *How to Report A Bug* section of the *Wine Users Guide* for information on how to write useful bug reports.

Crashes

These usually show up like this:

```
|Unexpected Windows program segfault - opcode = 8b
|Segmentation fault in Windows program 1b7:c41.
|Loading symbols from ELF file /root/wine/wine...
|...more Loading symbols from ...
|In 16 bit mode.
|Register dump:
|  CS:01b7 SS:016f DS:0287 ES:0000
|  IP:0c41 SP:878a BP:8796 FLAGS:0246
|  AX:811e BX:0000 CX:0000 DX:0000 SI:0001 DI:ffff
|Stack dump:
|0x016f:0x878a:  0001 016f ffed 0000 0000 0287 890b 1e5b
|0x016f:0x879a:  01b7 0001 000d 1050 08b7 016f 0001 000d
|0x016f:0x87aa:  000a 0003 0004 0000 0007 0007 0190 0000
|0x016f:0x87ba:
|
|0050: sel=0287 base=40211d30 limit=0b93f (bytes) 16-bit rw-
|Backtrace:
|0 0x01b7:0x0c41 (PXSRV_FONGETFACENAME+0x7c)
|1 0x01b7:0x1e5b (PXSRV_FONPUTCATFONT+0x2cd)
|2 0x01a7:0x05aa
|3 0x01b7:0x0768 (PXSRV_FONINITFONTS+0x81)
|4 0x014f:0x03ed (PDOXWIN_@SQLCURCB$Q6CBTYPEEULN8CBSCSTYPE+0x1b1)
|5 0x013f:0x00ac
|
|0x01b7:0x0c41 (PXSRV_FONGETFACENAME+0x7c):  movw          %es:0x38(%bx),%dx
```

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

1. Get the reason for the crash. This is usually an access to an invalid selector, an access to an out of range address in a valid selector, popping a segment register from the stack or the like. When reporting a crash, report this *whole* crashdump even if it doesn't make sense to you.

(In this case it is access to an invalid selector, for %es is 0000, as seen in the register dump).

2. Determine the cause of the crash. Since this is usually a primary/secondary reaction to a failed or misbehaving Wine function, rerun Wine with the `WINEDEBUG=+relay` environment variable set. This will generate quite a lot of output, but usually the reason is located in the last call(s). Those lines usually look like this:

```
|Call  KERNEL.90: LSTRLEN(0227:0692 "text") ret=01e7:2ce7 ds=0227  
|      ^^^^^^^^   ^       ^^^^^^^^^^    ^^^^^^          ^^^^^^^^^^   ^^^  
|              |           |               |                |Return address  
|              |           |               | textual parameter  
|              |           |Argument(s). This one is a winl6 segmented pointer.  
|              |Function called.  
|The module, the function is called in. In this case it is KERNEL.
```



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

3. If you have found a misbehaving function, try to find out why it misbehaves. Find the function in the source code. Try to make sense of the arguments passed. Usually there is a `WINE_DEFAULT_DEBUG_CHANNEL(<channel>);` at the beginning of the file. Rerun wine with the `WINEDEBUG=+xyz,+relay` environment variable set.

Occasionally there are additional debug channels defined at the beginning of the file in the form. `WINE_DECLARE_DEBUG_CHANNEL(<channel>);` If so the offending function may also use one of these alternate channels. Look through the the function for `TRACE(<channel>)(" ... /n");` and add any additional channels to the commandline.

4. Additional information on how to debug using the internal debugger can be found in `programs/winedbg/README`.
5. If this information isn't clear enough or if you want to know more about what's happening in the function itself, try running wine with `WINEDEBUG=+all`, which dumps ALL included debug information in wine.
6. If even that isn't enough, add more debug output for yourself into the functions you find relevant. See The section on Debug Logging in this guide for more information. You might also try to run the program in **gdb** instead of using the Wine debugger. If you do that, use `handle SIGSEGV nostop noprint` to disable the handling of seg faults inside **gdb** (needed for Win16).
7. You can also set a breakpoint for that function. Start wine using **winedbg** instead of **wine**. Once the debugger is is running enter **break** `KERNEL_LSTRLEN` (replace by function you want to debug, CASE IS RELEVANT) to set a breakpoint. Then use **continue** to start normal program-execution. Wine will stop if it reaches the breakpoint. If the program isn't yet at the crashing call of that function, use **continue** again until you are about to enter that function. You may now proceed with single-stepping the function until you reach the point of crash. Use the other debugger commands to print registers and the like.

Program hangs, nothing happens

Start the program with **winedbg** instead of **wine**. When the program locks up switch to the **winedbg** terminal and press **Ctrl-C**. this will stop the program and let you debug the program as you would for a crash.

Program reports an error with a MessageBox

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

Since the failure happens usually directly before setting up the MessageBox you can start winepdb and set a breakpoint at `MessageBoxA` (called by `win16` and `win32` programs) and proceed with **continue**. With `WINEDEBUG=+all` Wine will now stop directly before setting up the MessageBox. Proceed as explained above.

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

Disassembling programs:

You may also try to disassemble the offending program to check for undocumented features and/or use of them.

The best, freely available, disassembler for Win16 programs is Windows Codeback, archive name `wcbxxx.zip` (e.g. `wcb105a.zip`), which usually can be found in the Cica-Mirror subdirectory on the Wine ftp sites. (See ANNOUNCE).

Disassembling win32 programs is possible using Windows Disassembler 32. Look for a file called `w32dsm87.zip` (or similar) on <http://www.winsite.com>¹ and mirrors. The shareware version does not allow saving of disassembly listings. You can also use the newer (and in the full version better) Interactive Disassembler (IDA) from the ftp sites mentioned at the end of the document. Understanding disassembled code is mostly a question of exercise.

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

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

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

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

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

This does mostly the same as above, but also adds 0x86 bytes of stackstorage, which is accessed using `[bp-xx]`. Before calling a function, arguments are pushed on the stack using something like this:

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

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

Sample debugging session:

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

```

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

marcus@jet $ WINEDEBUG+=relay,-debug wine winword.exe
CallTo32(wndproc=0x40065bc0,hwnd=000001ac,msg=00000081,wp=00000000,lp=00000000)
Win16 task 'winword': Breakpoint 1 at 0x01d7:0x001a
CallTo16(func=0127:0070,ds=0927)
Call WPROCS.24: TASK_RESCHEDULE() ret=00b7:1456 ds=0927
Ret WPROCS.24: TASK_RESCHEDULE() retval=0x8672 ret=00b7:1456 ds=0927
CallTo16(func=01d7:001a,ds=0927)
AX=0000 BX=3cb4 CX=1f40 DX=0000 SI=0000 DI=0927 BP=0000 ES=11f7
Loading symbols: /home/marcus/wine/wine...
Stopped on breakpoint 1 at 0x01d7:0x001a
In 16 bit mode.
Wine-dbg>break MessageBoxA <---- Set Breakpoint
Breakpoint 2 at 0x40189100 (MessageBoxA [msgbox.c:190])
Wine-dbg>c <---- Continue
Call KERNEL.91: INITTASK() ret=0157:0022 ds=08a7
AX=0000 BX=3cb4 CX=1f40 DX=0000 SI=0000 DI=08a7 ES=11d7 EFL=00000286
CallTo16(func=090f:085c,ds=0dcf,0x0000,0x0000,0x0000,0x0000,0x0800,0x0000,0x0000,0x0dcf)
... <----- Much debugoutput
Call KERNEL.136: GETDRIVETYPE(0x0000) ret=060f:097b ds=0927
^^^^^^ Drive 0 (A:)
Ret KERNEL.136: GETDRIVETYPE() retval=0x0002 ret=060f:097b ds=0927
^^^^^^ DRIVE_REMOVEABLE
(It is a floppy diskdrive.)

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

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

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

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

```

Whoops, it even detects that something is wrong!

```

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

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

```

```
OF_CREATE|OF_SHARE_EXCLUSIVE|OF_READWRITE
```

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

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

```
|Call USER.1: MESSAGEBOX(0x0000,0x09278376"You must close Windows and load SHARE.EXE be
```

And MessageBox'ed.

```
|Stopped on breakpoint 2 at 0x40189100 (MessageBoxA [msgbox.c:190])
|190      { <- the sourceline
In 32 bit mode.
Wine-dbg>
```

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

Debugging Tips

Here are some additional debugging tips:

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

```
WINEDEBUG=+relay wine program
```

to get a listing of the functions the program calls in its start function. Now you do a

```
winedbg winfile.exe
```

This way, you get into **winedbg**. Now you can set a breakpoint on any function the program calls in the start function and just type **c** to bypass the eventual calls of Winfile to this function until you are finally at the place where this function gets called by the crashing start function. Now you can proceed with your debugging as usual.

- If you try to run a program and it quits after showing an error messagebox, the problem can usually be identified in the return value of one of the functions executed before `MessageBox()`. That's why you should re-run the program with e.g.

```
WINEDEBUG=+relay wine <program name> &>relmsg
```

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

```
Call USER.1: MESSAGEBOX(0x0000,0x01ff1246 "Runtime error 219 at 0004:1056.",0x00000000
```

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

```
Call KERNEL.96: FREELIBRARY(0x0347) ret=01cf:1033 ds=01ff
CallTo16(func=033f:0072,ds=01ff,0x0000)
Ret  KERNEL.96: FREELIBRARY() retval=0x0001 ret=01cf:1033 ds=01ff
Call KERNEL.96: FREELIBRARY(0x036f) ret=01cf:1043 ds=01ff
```

```

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

```

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

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

```

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

```

Provided that segment `0x0004` is indeed segment `0x1cf`, we now we can use IDA (available at <http://www.filelibrary.com:8080/cgi-bin/freedownload/DOS/h/72/ida35bx.zip>²) to disassemble the part that caused the error. We just have to find the address of the call to `FreeLibrary()`. Some lines before that the runtime error occurred. But be careful! In some cases you don't have to disassemble the main program, but instead some DLL called by it in order to find the correct place where the runtime error occurred. That can be determined by finding the origin of the segment value (in this case `0x1cf`).

- If you have created a relay file of some crashing program and want to set a breakpoint at a certain location which is not yet available as the program loads the breakpoint's segment during execution, you may set a breakpoint to `GetVersion16/32` as those functions are called very often.

Then do a `c` until you are able to set this breakpoint without error message.

- Some useful programs:

IDA: <http://www.filelibrary.com:8080/cgi-bin/freedownload/DOS/h/72/ida35bx.zip>³

Very good DOS disassembler ! It's badly needed for debugging Wine sometimes.

XRAY: <http://garbo.uwasa.fi/pub/pc/sysinfo/xray15.zip>⁴

Traces DOS calls (Int 21h, DPML, ...). Use it with Windows to correct file management problems etc.

pedump: <ftp://ftp.simtel.net/pub/simtelnet/win95/prog/pedump.zip>⁵

Dumps the imports and exports of a PE (Portable Executable) DLL.

winedump:

Dumps the imports and exports of a PE (Portable Executable) DLL (included in wine tree).

Some basic debugger usages:

After starting your program with

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

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

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

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

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

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

Useful memory addresses

Wine uses several different kinds of memory addresses.

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

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

Win16 "enhanced mode": segmented addresses.

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

Example:

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

DOS/Win16 "standard mode"

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

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

Configuration

Registry configuration

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

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

Determine the behavior:

Debugger:

this is the command line used to launch the debugger (it uses two `printf` formats (%ld) to pass context dependent information to the debugger). You should put here a complete path to your debugger (**WineDbg** can of course be used, but any other Windows' debugging API aware debugger will do). The path to the debugger you chose to use must be reachable via a DOS drive in the Wine config file !

Auto:

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

A regular Wine registry looks like:

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

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

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

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

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

WineDbg configuration

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

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

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

```
set $BreakAllThreadsStartup = 1
```

sets the option `BreakAllThreadsStartup` to `TRUE`.

All the options are read from the registry when **WineDbg** starts (if no corresponding value is found, a default value is used), and are written back to the registry when **WineDbg** exits (hence, all modifications to those options are automatically saved when **WineDbg** terminates).

Here's the list of all options:

Controlling when the debugger is entered

`BreakAllThreadsStartup`

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

`BreakOnCritSectTimeOut`

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

`BreakOnAttach`

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

BreakOnFirstChance

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

BreakOnDllLoad

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

Context information

ThreadId

ID of the `W`-thread currently examined by the debugger

ProcessId

ID of the `W`-thread currently examined by the debugger

<registers>

All CPU registers are also available

The `ThreadId` and `ProcessId` variables can be handy to set conditional breakpoints on a given thread or process.

WineDbg Command Reference

Misc

`abort` aborts the debugger

`quit` exits the debugger

`attach N` attach to a `W`-process (`N` is its ID, numeric or hexadecimal(`0xN`)).

IDs can be obtained using the `walk process` command. Note the

`walk process` command returns hexadecimal values

`detach` detach from a `W`-process. `WineDbg` will exit (this may be changed later on)

`help` prints some help on the commands

`help info` prints some help on info commands

`mode 16` switch to 16 bit mode

`mode 32` switch to 32 bit mode

Flow control

```

cont  continue execution until next breakpoint or exception.
pass  pass the exception event up to the filter chain.
step  continue execution until next C line of code (enters
      function call)
next  continue execution until next C line of code (doesn't
      enter function call)
stepi execute next assembly instruction (enters function
      call)
nexti execute next assembly instruction (doesn't enter
      function call)
finish do nexti commands until current function is exited

```

cont, step, next, stepi, nexti can be postfixed by a number (N), meaning that the command must be executed N times.

Breakpoints, watch points

```

enable N enables (break|watch)point #N
disable N disables (break|watch)point #N
delete N deletes (break|watch)point #N
cond N removes any a existing condition to (break|watch)point N
cond N <expr> adds condition <expr> to (break|watch)point N. <expr>
      will be evaluated each time the breakpoint is hit. If
      the result is a zero value, the breakpoint isn't
      triggered
break * N adds a breakpoint at address N
break <id> adds a breakpoint at the address of symbol <id>
break <id> N adds a breakpoint at the address of symbol <id> (N ?)
break N adds a breakpoint at line N of current source file
break adds a breakpoint at current $pc address
watch * N adds a watch command (on write) at address N (on 4 bytes)
watch <id> adds a watch command (on write) at the address of
      symbol <id>
info break lists all (break|watch)points (with state)

```

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

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

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

When setting a break/watch-point by <id>, if the symbol cannot be found (for example, the symbol is contained in a not yet loaded module), wineDBG will recall the name of the symbol and will try to set the breakpoint each time a new module is loaded (until it succeeds).

Stack manipulation

```

bt  print calling stack of current thread
bt N print calling stack of thread of ID N (note: this
      doesn't change the position of the current frame as
      manipulated by the up & dn commands)
up  goes up one frame in current thread's stack
up N goes up N frames in current thread's stack
dn  goes down one frame in current thread's stack
dn N goes down N frames in current thread's stack

```

```
frame N set N as the current frame for current thread's stack
info local prints information on local variables for current
function
```

Directory & source file manipulation

```
show dir
dir <pathname>
dir
symbolfile <pathname> loads external symbol definition
symbolfile <pathname> N loads external symbol definition
                        (applying an offset of N to addresses)

list lists 10 source lines from current position
list - lists 10 source lines before current position
list N lists 10 source lines from line N in current file
list <path>:N lists 10 source lines from line N in file <path>
list <id> lists 10 source lines of function <id>
list * N lists 10 source lines from address N
```

You can specify the end target (to change the 10 lines value) using the `''`. For example:

```
list 123, 234 lists source lines from line 123 up to line 234 in
current file
list foo.c:1,56 lists source lines from line 1 up to 56 in file foo.c
```

Displaying

A display is an expression that's evaluated and printed after the execution of any **WineDbg** command.

```
display lists the active displays
info display (same as above command)
display <expr> adds a display for expression <expr>
display /fmt <expr> adds a display for expression <expr>. Printing
evaluated <expr> is done using the given format (see
print command for more on formats)
del display N deletes display #N
undisplay N (same as del display)
```

Disassembly

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

Information on Wine's internals

```

info class <id> prints information on Windows's class <id>
walk class lists all Windows' class registered in Wine
info share lists all the dynamic libraries loaded the debugged
    program (including .so files, NE and PE DLLs)
info module <N> prints information on module of handle <N>
walk module lists all modules loaded by debugged program
info regs prints the value of CPU register
info segment <N> prints information on segment <N>
info segment lists all allocated segments
info stack prints the values on top of the stack
walk map lists all virtual mappings used by the debugged
    program
walk map <N>    lists all virtual mappings used by the program of pid <N>
info wnd <N>    prints information of Window of handle <N>
walk wnd lists all the window hierarchy starting from the
    desktop window
walk wnd <N>    lists all the window hierarchy starting from the
    window of handle <N>
walk process lists all w-processes in Wine session
walk thread lists all w-threads in Wine session
walk exception lists the exception frames (starting from current
    stack frame)

```

Memory (reading, writing, typing)

```

x <expr> examines memory at <expr> address
x /fmt <expr> examines memory at <expr> address using format /fmt
print <expr> prints the value of <expr> (possibly using its type)
print /fmt <expr> prints the value of <expr> (possibly using its
    type)
set <lval>=<expr> writes the value of <expr> in <lval>
whatis <expr> prints the C type of expression <expr>

```

/fmt is either /<letter> or /<count><letter> letter can be

```

s => an ASCII string
u => an Unicode UTF16 string
i => instructions (disassemble)
x => 32 bit unsigned hexadecimal integer
d => 32 bit signed decimal integer
w => 16 bit unsigned hexadecimal integer
c => character (only printable 0x20-0x7f are actually printed)
b => 8 bit unsigned hexadecimal integer
g => GUID

```

Expressions

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

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

Debug channels

It is possible to turn on and off debug messages as you are debugging using the `set` command.

```
set + warn win => turn on warn on 'win' channel
set + win => turn on warn/fixme/err/trace on 'win' channel
set - win => turn off warn/fixme/err/trace on 'win' channel
set - fixme => turn off the 'fixme' class
```

Other debuggers

GDB mode

WineDbg can act as a remote monitor for GDB. This allows to use all the power of GDB, but while debugging wine and/or any Win32 application. To enable this mode, just add `--gdb` to `winedbg` command line. You'll end up on a GDB prompt. You'll have to use the GDB commands (not the wine nes).

However, some limitation in GDB while debugging wine (see below) don't appear in this mode:

- GDB will correctly present Win32 thread information and breakpoint behavior
- Moreover, it also provides support for the Dwarf II debug format (which became the default format (instead of stabs) in gcc 3.1).

A few wine extensions available through the monitor command.

<code>monitor wnd</code>	lists all window in the Wine session
<code>monitor proc</code>	lists all processes in the Wine session
<code>monitor mem</code>	displays memory mapping of debugged process (doesn't work)

Graphical frontends to gdb

This section will describe how you can debug wine using the GDB mode of `winedbg` and some graphical front ends to GDB for those of you who really like graphical debuggers.

DDD

Use the following steps, in this order:

1. Start the wine debugger with a command line like:

```
winedbg -- --gdb --no-start <name_of_exe_to_debug.exe>
```

2. Start ddd
3. In ddd, use the 'Open File' or 'Open Program' to point to the wine executable

4. In the output of 1/, there's a line like

```
target remote localhost:32878
```

copy that line and paste into ddd command pane (the one with the (gdb) prompt)

The program should now be loaded and up and running. If you want, you can also add in 1/ after the name of the exec all the needed parameters

kdbg

Use the following steps, in this order:

1. Start the wine debugger with a command line like:

```
winedbg -- --gdb --no-start <name_of_exe_to_debug.exe>
```

2. In the output of 1/, there's a line like

```
target remote localhost:32878
```

Start kdbg with

```
kdbg -r localhost:32878 wine
```

localhost:32878 is not a fixed value, but has been printed in step 1/. 'wine' should also be the full path to the wine executable.

The program should now be loaded and up and running. If you want, you can also add in 1/ after the name of the exec all the needed parameters

Using other Unix debuggers

You can also use other debuggers (like **gdb**), but you must be aware of a few items:

You need to attach the unix debugger to the correct unix process (representing the correct windows thread) (you can "guess" it from a **ps fax** for example: When running the emulator, usually the first two **upids** are for the Windows' application running the desktop, the first thread of the application is generally the third **upid**; when running a Winelib program, the first thread of the application is generally the first **upid**)

Note: Even if latest **gdb** implements the notion of threads, it won't work with Wine because the thread abstraction used for implementing Windows' thread is not 100% mapped onto the Linux POSIX threads implementation. It means that you'll have to spawn a different **gdb** session for each Windows' thread you wish to debug.

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

Change into your Wine source dir and enter:

```
$ gdb wine
```

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

```
(gdb) attach PID
```

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

```
(gdb) bt
```

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

```
(gdb) n
```

or maybe even

```
(gdb) b SomeFunction
```

and

```
(gdb) c
```

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

```
(gdb) detach
```

to detach from the Wine process.

Using other Windows debuggers

You can use any Windows' debugging API compliant debugger with Wine. Some reports have been made of success with VisualStudio debugger (in remote mode, only the hub runs in Wine). GoVest fully runs in Wine.

Main differences between wineDbg and regular Unix debuggers

Table 1-1. Debuggers comparison

WineDbg	gdb
WineDbg debugs a Windows' process: the various threads will be handled by the same WineDbg session, and a breakpoint will be triggered for any thread of the W-process	gdb debugs a Windows' thread: a separate gdb session is needed for each thread of a Windows' process and a breakpoint will be triggered only for the w-thread debugged
WineDbg supports debug information from stabs (standard Unix format) and Microsoft's C, CodeView, .DBG	GDB supports debug information from stabs (standard Unix format) and Dwarf II.

Limitations

- 16 bit processes are not supported (but calls to 16 bit code in 32 bit applications are).
- Function call in expression is no longer supported

Notes

1. <http://www.winsite.com/>
2. <http://www.filelibrary.com:8080/cgi-bin/freedownload/DOS/h/72/ida35bx.zip>
3. <http://www.filelibrary.com:8080/cgi-bin/freedownload/DOS/h/72/ida35bx.zip>
4. <http://garbo.uwasa.fi/pub/pc/sysinfo/xray15.zip>
5. <ftp://ftp.simtel.net/pub/simtelnet/win95/prog/pedump.zip>

Chapter 2. Documenting Wine

This chapter describes how you can help improve Wine's documentation.

Like most large scale volunteer projects, Wine is strongest in areas that are rewarding for its volunteers to work in. The majority of contributors send code patches either fixing bugs, adding new functionality or otherwise improving the software components of the distribution. A lesser number contribute in other ways, such as reporting bugs and regressions, creating tests, providing organizational assistance, or helping to document Wine.

Documentation is important for many reasons, and is often the key to the end user having a successful experience in installing, setting up and using software. Because Wine is a complicated, evolving entity, providing quality up to date documentation is vital to encourage more people to persevere with using and contributing to the project. The following sections describe in detail how to go about adding to or updating Wine's existing documentation.

An Overview Of Wine Documentation

The Wine source code tree comes with a large amount of documentation in the `documentation/` subdirectory. This used to be a collection of text files culled from various places such as the Wine Weekly News and the wine-devel mailing list, but was reorganized some time ago into a number of books, each of which is marked up using SGML. You are reading one of these books (the *Wine Developer's Guide*) right now.

Since being reorganized, the books have been updated and extended regularly. In their current state they provide a good framework which over time can be expanded and kept up to date. This means that most of the time when further documentation is added, it is a simple matter of updating the content of an already existing file. The books available at the time of writing are:

- The *Wine User Guide*. This book contains information for end users on installing, configuring and running Wine.
- The *Wine Developer's Guide*. This book contains information and guidelines for developers and contributors to the Wine project.
- The *Wine User's Guide*. This book contains information for developers using Wine to port Win32 applications to Unix.
- The *Wine Packager's Guide*. This book contains information for anyone who will be distributing Wine to end users in a prepackaged format. It is also the exception to the rule as it has intentionally been kept in text format.
- The *Wine FAQ*. This book contains frequently asked questions about Wine with their answers.

Another source of documentation is the *Wine API Guide*. This is generated information taken from special comments placed in the Wine source code. When you update or add new API calls to Wine you should consider documenting them so that developers can determine what the API does and how it should be used.

The next sections describe how to create Wine API documentation and how to work with SGML so you can add to the existing books.

Writing Wine API Documentation

Introduction to API Documentation

Wine includes a large amount of documentation on the API functions it implements. There are several reasons to want to document the Win32 API:

- To allow Wine developers to know what each function should do, should they need to update or fix it.
- To allow Winelib users to understand the functions that are available to their applications.
- To provide an alternative source of free documentation on the Win32 API.
- To provide more accurate documentation where the existing documentation is accidentally or deliberately vague or misleading.

To this end, a semi formalized way of producing documentation from the Wine source code has evolved. Since the primary users of API documentation are Wine developers themselves, documentation is usually inserted into the source code in the form of comments and notes. Good things to include in the documentation of a function include:

- The purpose of the function.
- The parameters of the function and their purpose.
- The return value of the function, in success as well as failure cases.
- Additional notes such as interaction with other parts of the system, differences between Wine's implementation and Win32s, errors in MSDN documentation, undocumented cases and bugs that Wine corrects or is compatible with.

Good documentation helps developers be aware of the effects of making changes. It also allows good tests to be written which cover all of the documented cases.

Note that you do not need to be a programmer to update the documentation in Wine. If you would like to contribute to the project, patches that improve the API documentation are welcome. The following describes how to format any documentation that you write so that the Wine documentation generator can extract it and make it available to other developers and users.

In general, if you did not write the function in question, you should be wary of adding comments to other peoples code. It is quite possible you may misunderstand or misrepresent what the original author intended! Adding API documentation on the other hand can be done by anybody, since in most cases there is plenty of information about what a function is supposed to do (if it isn't obvious) available in books and articles on the internet.

A final warning concerns copyright and must be noted. If you read MSDN or any publication in order to find out what an API call does, you must be aware that the text you are reading is copyrighted and in most cases cannot legally be reproduced without the authors permission. If you copy verbatim any information from such sources and submit it for inclusion into Wine, you open yourself up to potential legal liability. You must ensure that anything you submit is your own work, although it can be based on your understanding gleaned from reading other peoples work.

Basic API Documentation

The general form of an API comment in Wine is a block comment immediately before a function is implemented in the source code. General comments within a function body or at the top of an implementation file are ignored by the API documentation generator. Such comments are for the benefit of developers only, for example to explain what the source code is doing or to describe something that may not be obvious to the person reading the source code.

The following text uses the function *PathRelativePathToA()* from *SHLWAPI.DLL* as an example. You can find this function in the Wine source code tree in the file *dlls/shlwapi/path.c*.

The first line of the comment gives the name of the function, the DLL that the function is exported from, and its export ordinal number. This is the simplest (and most common type of) comment:

```

/*****
 * PathRelativePathToA    [SHLWAPI.@]
 */

```

The functions name and the DLL name are obvious. The ordinal number takes one of two forms: Either *@* as in the above, or a number if the export is exported by ordinal. You can see which to use by looking at the DLL's *.spec* file. If the line on which the function is listed begins with a number, use it, otherwise use the *@* symbol, which indicates that this function is imported only by name.

Note also that round or square brackets can be used, and whitespace between the name and the DLL/ordinal is free form. Thus the following is equally valid:

```

/*****
 * PathRelativePathToA (SHLWAPI.@)
 */

```

This basic comment will not get processed into documentation, since it contains no information. In order to produce documentation for the function, We must add some of the information listed above.

First we add a description of the function. This can be as long as you like, but typically contains only a brief description of what the function is meant to do in general terms. It is free form text:

```

/*****
 * PathRelativePathToA    [SHLWAPI.@]
 *
 * Create a relative path from one path to another.
 */

```

To be truly useful however we must document the parameters to the function. There are two methods for doing this: In the comment, or in the function prototype.

Parameters documented in the comment should be formatted as follows:

```

/*****
 * PathRelativePathToA    [SHLWAPI.@]
 *
 * Create a relative path from one path to another.
 *
 * PARAMS
 *   lpszPath    [O] Destination for relative path
 *   lpszFrom    [I] Source path
 *   dwAttrFrom [I] File attribute of source path
 *   lpszTo      [I] Destination path
 */

```

```

*   dwAttrTo    [I] File attributes of destination path
*
*/

```

The parameters section starts with **PARAMS** on its own line. Each parameter is listed in the order they appear in the functions prototype, first with the parameters name, followed by its input/output status, followed by a free form text description of the comment.

The input/output status tells the programmer whether the value will be modified by the function (an output parameter), or only read (an input parameter). The status must be enclosed in square brackets to be recognized, otherwise, or if it is absent, anything following the parameter name is treated as the parameter description. This field is case insensitive and can be any of the following: **[I]**, **[In]**, **[O]**, **[Out]**, **[I/O]**, **[In/Out]**.

Parameters documented in the prototype should be formatted as follows:

```

/*****
*   PathRelativePathToA    [SHLWAPI.@]
*
*   Create a relative path from one path to another.
*
*/
BOOL WINAPI PathRelativePathToA(
    LPSTR  lpszPath,    /* [O] Destination for relative path */
    LPCSTR lpszFrom,    /* [I] Source path */
    DWORD  dwAttrFrom, /* [I] File attribute of source path */
    LPCSTR lpszTo,      /* [I] Destination path */
    DWORD  dwAttrTo)    /* [I] File attributes of destination path */

```

The choice of which style to use is up to you, although for readability it is suggested you stick with the same style within a single source file.

Following the description and parameters come a number of optional sections, all in the same format. A section is defined as the section name, which is an all upper case section name on its own line, followed by free form text. You can create any sections you like, however for consistency it is recommended you use the following section names:

1. **NOTES.** Anything that needs to be noted about the function such as special cases and the effects of input arguments.
2. **BUGS.** Any bugs in the function that exist 'by design', i.e. those that will not be fixed or exist for compatibility with Windows.
3. **TODO.** Any unhandled cases or missing functionality in the Wine implementation of the function.
4. **FIXME.** Things that should be updated or addressed in the implementation of the function at some future date (perhaps dependent on other parts of Wine). Note that if this information is only relevant to Wine developers then it should probably be placed in the relevant code section instead.

Following or before the optional sections comes the **RETURNS** section which describes the return value of the function. This is free form text but should include what is returned on success as well as possible error return codes. Note that this section must be present for documentation to be generated for your comment.

Our final documentation looks like the following:

```

/*****

```

```

* PathRelativePathToA    [SHLWAPI.@]
*
* Create a relative path from one path to another.
*
* PARAMS
*   lpszPath    [O] Destination for relative path
*   lpszFrom    [I] Source path
*   dwAttrFrom  [I] File attribute of source path
*   lpszTo      [I] Destination path
*   dwAttrTo    [I] File attributes of destination path
*
* RETURNS
*   TRUE  If a relative path can be formed. lpszPath contains the new path
*   FALSE If the paths are not relative or any parameters are invalid
*
* NOTES
*   lpszTo should be at least MAX_PATH in length.
*   Calling this function with relative paths for lpszFrom or lpszTo may
*   give erroneous results.
*
*   The Win32 version of this function contains a bug where the lpszTo string
*   may be referenced 1 byte beyond the end of the string. As a result random
*   garbage may be written to the output path, depending on what lies beyond
*   the last byte of the string. This bug occurs because of the behaviour of
*   PathCommonPrefix() (see notes for that function), and no workaround seems
*   possible with Win32.
*   This bug has been fixed here, so for example the relative path from "\\\"
*   to "\\\" is correctly determined as "." in this implementation.
*/

```

Advanced API Documentation

There is no markup language for formatting API comments, since they should be easily readable by any developer working on the source file. A number of constructs are treated specially however, and are noted here. You can use these constructs to enhance the usefulness of the generated documentation by making it easier to read and referencing related documents.

Any valid `c` identifier that ends with `()` is taken to be an API function and is formatted accordingly. When generating documentation, this text will become a link to that API call, if the output type supports hyperlinks or their equivalent.

Similarly, any interface name starting with a capital `I` and followed by the words "reference" or "object" become a link to that objects documentation.

Where an Ascii and Unicode version of a function are available, it is recommended that you document only the Ascii version and have the Unicode version refer to the Ascii one, as follows:

```

/*****
* PathRelativePathToW    [SHLWAPI.@]
*
* See PathRelativePathToA.
*/

```

Alternately you may use the following form:

```

/*****
* PathRelativePathToW    [SHLWAPI.@]
*
* Unicode version of PathRelativePathToA.
*/

```

You may also use this construct in any other section, such as **NOTES**.

Any numbers and text in quotes (""") are highlighted.

Words in all uppercase are assumed to be API constants and are highlighted. If you want to emphasize something in the documentation, put it in a section by itself rather than making it upper case.

Blank lines in a section cause a new paragraph to be started. Blank lines at the start and end of sections are ignored.

Any comment line starting with ("* |") is treated as raw text and is not pre-processed before being output. This should be used for code listings, tables and any text that should remain unformatted.

Any line starting with a single word followed by a colon (:) is assumed to be case listing and is emphasized and put in its own paragraph. This is most often used for return values, as in the example section below.

```
* RETURNS
* Success: TRUE. Something happens that is documented here.
* Failure: FALSE. The reasons why this call can fail are listed here.
```

Any line starting with a (-) is put into a paragraph by itself. this allows lists to avoid being run together.

If you are in doubt as to how your comment will look, try generating the API documentation and checking the output.

Extra API Documentation

Simply documenting the API calls available provides a great deal of information to developers working with the Win32 API. However additional documentation is needed before the API Guide can be considered truly useful or comprehensive. For example, COM objects that are available for developers use should be documented, along with the interface(s) that those objects export. Also, it would be helpful to document each dll, to provide some structure to the documentation.

To facilitate providing extra documentation, you can create comments that provide extra documentation on functions, or on keywords such as the name of a COM interface or a type definition.

These items are generated using the same formatting rules as described earlier. The only difference is the first line of the comment, which indicates to the generator that the documentation is supplemental and does not describe an export from the dll being processed.

Lets assume you have implemented a COM interface that you want to document; we'll use the name **IExample** as an example here. Your comment would look like the following (assuming you are exporting this object from `EXAMPLE.DLL`):

```
/* *****
 * IExample    {EXAMPLE}
 *
 * The IExample object provides lots of interesting functionality.
 * ...
 */
```

Format this documentation exactly as you would a standard export. The only difference is the use of curly brackets to mark this documentation as supplemental.

The generator will output this documentation using the name given before the DLL name, and will link to it from the main DLL page. In addition, if you have referred to the comment name in other documentation using "IExample interface", "IExample object", or "IExample()", those references will point to this documentation.

If you document you COM interfaces this way then all following extra comments that follow in the same source file that begin with the same document title will be added as references to this comment before it is output. For an example of this see `dlls/oleaut32/safearray.c`. This uses an extra comment to document The SafeArray functions and link them together under one heading.

As a special case, if you use the DLL name as the comment name, the comment will be treated as documentation on the DLL itself. When the documentation for the DLL is processed, the contents of the comment will be placed before the generated statistics, exports and other information that makes up a DLL's documentation page.

Generating API Documentation

Having edited or added new API documentation to a source code file, you should generate the documentation to ensure that the result is what you expected. Wine includes a tool (slightly misleadingly) called **c2man.pl** in the `tools/` directory which is used to generate the documentation from the source code.

You can run **c2man.pl** manually for testing purposes; it is a fairly simple perl script which parses `.c` files to create output in several formats. If you wish to try this you may want to run it with no arguments, which will cause it to print usage information.

An easier way is to use Wine's build system. To create man pages for a given dll, just type **make man** from within the `dlls` directory or type **make manpages** in the root directory of the Wine source tree. You can then check that a man page was generated for your function, it should be present in the `documentation/man3w` directory with the same name as the function.

Once you have generated the man pages from the source code, running **make install** will install them for you. By default they are installed in section 3w of the manual, so they don't conflict with any existing man page names. So, to read the man page you should use **man -S 3w {name}**. Alternately you can edit `/etc/man.config` and add 3w to the list of search paths given in the variable `MANSECT`.

You can also generate HTML output for the API documentation, in this case the make command is **make doc-html** in the `dll` directory, or **make htmlpages** from the root. The output will be placed by default under `documentation/html`. Similarly you can create SGML source code to produce the *Wine Api Guide* with the command **make sgmlpages**.

The Wine DocBook System

Writing Documentation with DocBook

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

Getting Started

Why SGML?: The simple answer to that is that SGML allows you to create multiple formats of a given document from a single source. Currently it is used to create HTML, PDF, PS (PostScript) and Text versions of the Wine books.

What do I need?: You need the SGML tools. There are various places where you can get them. The most generic way of getting them is from their source as discussed below.

Quick instructions: These are the basic steps to create the Wine books from the SGML source.

1. Go to <http://www.sgmltools.org>
2. Download all of the sgmltools packages
3. Install them all and build them (**`./configure; make; make install`**)
4. Switch to your toplevel Wine directory
5. Run **`./configure`** (or **`make distclean && ./configure`**)
6. Switch to the `documentation/` directory
7. run **`make html`**
8. View `wine-user.html`, `wine-devel.html`, etc. in your favorite browser

Getting SGML for various distributions

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

SGML on Red Hat

The following packages seem to be sufficient for Red Hat 7.1. You will want to be careful about the order in which you install the RPMs.

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

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

SGML on Debian

This is not a definitive list yet, but it seems you might need the following packages:

- docbook
- docbook-dsssl
- docbook-utils
- docbook-xml
- docbook-xsl
- sgml-base
- sgml-data
- tetex-base
- tetex-bin
- jade
- jadetex

Terminology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Document

Once you're comfortable with SGML, creating a DocBook document is quite simple and straightforward. Even though DocBook contains over 300 different tags, you can usually get by with only a small subset of those tags. Most of them are for inline for-

matting, rather than for document structuring. Furthermore, the common tags have short, intuitive names.

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

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

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

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

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

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

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

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

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

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

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

```
</chapter>  
</book>
```

Common Elements

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

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

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

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

Here are the common SGML elements:

Structural Elements

`<book>`

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

`<set>`

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

`<chapter>`

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

`<part>`

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

`<sect?>`

DocBook has many section elements to divide the contents of a chapter into smaller chunks. The encouraged approach is to use the numbered section tags, `<sect1>`, `<sect2>`, `<sect3>`, `<sect4>`, and `<sect5>` (if necessary). These tags

must be nested in order: you can't place a `<sect3>` directly inside a `<sect1>`. You have to nest the `<sect3>` inside a `<sect2>`, and so forth. Documents with these explicit section groupings are easier for SGML processors to deal with, and lead to better organized documents. DocBook also supplies a `<section>` element which you can nest inside itself, but its use is discouraged in favor of the numbered section tags.

`<title>`

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

`<para>`

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

`<article>`

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

Inline Formatting Elements

`<filename>`

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

`<userinput>`

Literal text entered by the user.

`<computeroutput>`

Literal text output by the computer.

`<literal>`

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

`<quote>`

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

`<note>`

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

```
<note>
  <title>EXAMPLE</title>
  <para>This is an example note...</para>
</note>
```

The results will look something like this:

EXAMPLE: This is an example note...

`<sgmltag>`

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

`<prompt>`

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

`<replaceable>`

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

`<constant>`

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

`<symbol>`

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

`<function>`

A programming function name.

`<parameter>`

Programming language parameters you pass with a function.

`<option>`

Parameters you pass to a command-line executable.

`<varname>`

Variable name, typically in a programming language.

`<type>`

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

`<structname>`

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

`<structfield>`

A field inside a C struct.

`<command>`

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

`<envvar>`

An environment variable, e.g, \$PATH.

`<systemitem>`

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

`<email>`

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

`<firstterm>`

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

Item Listing Elements

`<itemizedlist>`

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

`<orderedlist>`

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

`<simplelist>`

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

`<variablelist>`

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

Block Text Quoting Elements

`<programlisting>`

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

`<screen>`

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

Hyperlink Elements

`<link>`

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

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

`<xref>`

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

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

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

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

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

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

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

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

`<anchor>`

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

`<ulink>`

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

`<olink>`

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

Editing SGML Documents

You can write SGML/DocBook documents in any text editor you might find although some editors are more friendly for this task than others.

The most commonly used open source SGML editor is Emacs, with the PSGML *mode*, or extension. Emacs does not supply a GUI or WYSIWYG (What You See Is What You Get) interface, but it does provide many helpful shortcuts for creating SGML, as well

as automatic formatting, validity checking, and the ability to create your own macros to simplify complex, repetitive actions.

Notes

1. <http://www.sgmltools.org>
2. <http://docbook.org>
3. <http://www.winehq.org>

Chapter 3. Submitting Patches

Patch Format

Patches are submitted via email to the Wine patches mailing list, <wine-patches@winehq.org>. Your patch should include:

- A meaningful subject (very short description of patch)
- A long (paragraph) description of what was wrong and what is now better. (recommended)
- A change log entry (short description of what was changed).
- The patch in **diff -u** format

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

For additions simply include them by appending the **diff -u /dev/null /my/new/file** output of them to any **cv**s **diff -u** output you may have. Alternatively, use **diff -Nu olddir/ newdir/** in case of multiple new files to add.

For removals, clearly list the files in the description of the patch.

Since wine is constantly changing due to development it is strongly recommended that you use **cv**s for patches, if you cannot use **cv**s for some reason, you can submit patches against the latest tarball. To do this make a copy of the files that you will be modifying and **diff -u** against the old file. I.E.

```
diff -u file.old file.c > file.txt
```

Some notes about style

There are a few conventions that about coding style that have been adopted over the years of development. The rational for these “rules” is explained for each one.

- No HTML mail, since patches should be in-lined and HTML turns the patch into garbage. Also it is considered bad etiquette as it uglifies the message, and is not viewable by many of the subscribers.
- Only one change set per patch. Patches should address only one bug/problem at a time. If a lot of changes need to be made then it is preferred to break it into a series of patches. This makes it easier to find regressions.
- Tabs are not forbidden but discouraged. A tab is defined as 8 characters and the usual amount of indentation is 4 characters.
- C++ style comments are discouraged since some compilers choke on them.
- Commenting out a block of code is usually done by enclosing it in **#if 0 ... #endif** Statements. For example.

```
/* note about reason for commenting block */
#if 0
code
code /* comments */
code
#endif
```

The reason for using this method is that it does not require that you edit comments that may be inside the block of code.

- Patches should be in-lined (if you can configure your email client to not wrap lines), or attached as plain text attachments so they can be read inline. This may mean some more work for you. However it allows others to review your patch easily and decreases the chances of it being overlooked or forgotten.
- Code is usually limited to 80 columns. This helps prevent mailers mangling patches by line wrap. Also it generally makes code easier to read.

Inline attachments with Outlook Express

Outlook Express is notorious for mangling attachments. Giving the patch a `.txt` extension and attaching will solve the problem for most mailers including Outlook. Also, there is a way to enable Outlook Express send `.diff` attachments.

You need following two things to make it work.

1. Make sure that `.diff` files have `\r\n` line ends, because if OE detects that there is no `\r\n` line endings it switches to quoted-printable format attachments.
2. Using regedit add key "Content Type" with value "text/plain" to the `.diff` extension under `HKEY_CLASSES_ROOT` (same as for `.txt` extension). This tells OE to use Content-Type: text/plain instead of application/octet-stream.

Item #1 is important. After you hit "Send" button, go to "Outbox" and using "Properties" verify the message source to make sure that the mail has correct format. You might want to send several test emails to yourself too.

Alexandre's Bottom Line

"The basic rules are: no attachments, no mime crap, no line wrapping, a single patch per mail. Basically if I can't do `"cat raw_mail | patch -p0"` it's in the wrong format."

Quality Assurance

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

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

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

Chapter 4. Writing Conformance tests

Introduction

The Windows API follows no standard, it is itself a de facto standard, and deviations from that standard, even small ones, often cause applications to crash or misbehave in some way.

The question becomes, "How do we ensure compliance with that standard?" The answer is, "By using the API documentation available to us and backing that up with conformance tests." Furthermore, a conformance test suite is the most accurate (if not necessarily the most complete) form of API documentation and can be used to supplement the Windows API documentation.

Writing a conformance test suite for more than 10000 APIs is no small undertaking. Fortunately it can prove very useful to the development of Wine way before it is complete.

- The conformance test suite must run on Windows. This is necessary to provide a reasonable way to verify its accuracy. Furthermore the tests must pass successfully on all Windows platforms (tests not relevant to a given platform should be skipped).

A consequence of this is that the test suite will provide a great way to detect variations in the API between different Windows versions. For instance, this can provide insights into the differences between the, often undocumented, Win9x and NT Windows families.

However, one must remember that the goal of Wine is to run Windows applications on Linux, not to be a clone of any specific Windows version. So such variations must only be tested for when relevant to that goal.

- Writing conformance tests is also an easy way to discover bugs in Wine. Of course, before fixing the bugs discovered in this way, one must first make sure that the new tests do pass successfully on at least one Windows 9x and one Windows NT version.

Bugs discovered this way should also be easier to fix. Unlike some mysterious application crashes, when a conformance test fails, the expected behavior and APIs tested for are known thus greatly simplifying the diagnosis.

- To detect regressions. Simply running the test suite regularly in Wine turns it into a great tool to detect regressions. When a test fails, one immediately knows what was the expected behavior and which APIs are involved. Thus regressions caught this way should be detected earlier, because it is easy to run all tests on a regular basis, and easier to fix because of the reduced diagnosis work.
- Tests written in advance of the Wine development (possibly even by non Wine developers) can also simplify the work of the future implementer by making it easier for him to check the correctness of his code.
- Conformance tests will also come in handy when testing Wine on new (or not as widely used) architectures such as FreeBSD, Solaris x86 or even non-x86 systems. Even when the port does not involve any significant change in the thread management, exception handling or other low-level aspects of Wine, new architectures can expose subtle bugs that can be hard to diagnose when debugging regular (complex) applications.

What to test for?

The first thing to test for is the documented behavior of APIs and such as `CreateFile`. For instance one can create a file using a long pathname, check that the behavior is correct when the file already exists, try to open the file using the corresponding short pathname, convert the filename to Unicode and try to open it using `CreateFileW`, and all other things which are documented and that applications rely on.

While the testing framework is not specifically geared towards this type of tests, it is also possible to test the behavior of Windows messages. To do so, create a window, preferably a hidden one so that it does not steal the focus when running the tests, and send messages to that window or to controls in that window. Then, in the message procedure, check that you receive the expected messages and with the correct parameters.

For instance you could create an edit control and use `WM_SETTEXT` to set its contents, possibly check length restrictions, and verify the results using `WM_GETTEXT`. Similarly one could create a listbox and check the effect of `LB_DELETESTRING` on the list's number of items, selected items list, highlighted item, etc.

However, undocumented behavior should not be tested for unless there is an application that relies on this behavior, and in that case the test should mention that application, or unless one can strongly expect applications to rely on this behavior, typically APIs that return the required buffer size when the buffer pointer is `NULL`.

Running the tests in Wine

The simplest way to run the tests in Wine is to type 'make test' in the Wine sources top level directory. This will run all the Wine conformance tests.

The tests for a specific Wine library are located in a 'tests' directory in that library's directory. Each test is contained in a file (e.g. `dlls/kernel/tests/thread.c`). Each file itself contains many checks concerning one or more related APIs.

So to run all the tests related to a given Wine library, go to the corresponding 'tests' directory and type 'make test'. This will compile the tests, run them, and create an 'xxx.ok' file for each test that passes successfully. And if you only want to run the tests contained in the `thread.c` file of the kernel library, you would do:

```
$ cd dlls/kernel/tests
$ make thread.ok
```

Note that if the test has already been run and is up to date (i.e. if neither the kernel library nor the `thread.c` file has changed since the `thread.ok` file was created), then make will say so. To force the test to be re-run, delete the `thread.ok` file, and run the make command again.

You can also run tests manually using a command similar to the following:

```
$ ../../../../tools/runtest -q -M kernel32.dll -p kernel32_test.exe.so thread.c
$ ../../../../tools/runtest -p kernel32_test.exe.so thread.c
thread.c: 86 tests executed, 5 marked as todo, 0 failures.
```

The '-P wine' options defines the platform that is currently being tested. Remove the '-q' option if you want the testing framework to report statistics about the number of successful and failed tests. Run **runtest -h** for more details.

Cross-compiling the tests with MinGW

Setup of the MinGW cross-compiling environment

Here are some instructions to setup MinGW on different Linux distributions and *BSD.

Debian GNU/Linux

On Debian all you need to do is type **apt-get install mingw32**.

Red Hat Linux like rpm systems

This includes Fedora Core, Red Hat Enterprise Linux, Mandrake, most probably SuSE Linux too, etc. But this list isn't exhaustive; the following steps should probably work on any rpm based system.

Download and install the latest rpm's from MinGW RPM packages¹. Alternatively you can follow the instructions on that page and build your own packages from the source rpm's listed there as well.

*BSD

The *BSD systems have in their ports collection a port for the MinGW cross-compiling environment. Please see the documentation of your system about how to build and install a port.

Compiling the tests

Having the cross-compiling environment set up the generation of the Windows executables is easy by using the Wine build system.

If you had already run **configure**, then delete `config.cache` and re-run **configure**. You can then run **make crosstest**. To sum up:

```
$ rm config.cache
$ ./configure
$ make crosstest
```

Building and running the tests on Windows

Using pre-compiled binaries

Unfortunately there are no pre-compiled binaries yet. However if send an email to the Wine development list you can probably get someone to send them to you, and maybe motivate some kind soul to put in place a mechanism for publishing such binaries on a regular basis.

With Visual C++

Visual Studio 6 users:

- MSVC headers may not work well, try with Wine headers
- Ensure that you have the "processor pack" from <http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.aspx> as well as the latest service packs. The processor pack fixes "error C2520: conversion of `__int64` to `double` not implemented, use signed `__int64`"

- get the Wine sources
- Run `msvcmaker` to generate Visual C++ project files for the tests. 'msvcmaker' is a perl script so you may be able to run it on Windows.

```
$ ./tools/winapi/msvcmaker --no-wine
```

- If the previous steps were done on your Linux development machine, make the Wine sources accessible to the Windows machine on which you are going to compile them. Typically you would do this using Samba but copying them altogether would work too.
- On the Windows machine, open the `winetest.dsw` workspace. This will load each test's project. For each test there are two configurations: one compiles the test with the Wine headers, and the other uses the Visual C++ headers. Some tests will compile fine with the former, but most will require the latter.
- Open the **Build+Batch build...** menu and select the tests and build configurations you want to build. Then click on **Build**.
- To run a specific test from Visual C++, go to **Project+Settings....** There select that test's project and build configuration and go to the *Debug* tab. There type the name of the specific test to run (e.g. 'thread') in the *Program arguments* field. Validate your change by clicking on **Ok** and start the test by clicking the red exclamation mark (or hitting 'F5' or any other usual method).
- You can also run the tests from the command line. You will find them in either `Output\Win32_Wine-Headers` or `Output\Win32_MSVC-Headers` depending on the build method. So to run the kernel 'path' tests you would do:

```
C:\>cd dlls\kernel\tests\Output\Win32_MSVC-Headers
C:\dlls\kernel\tests\Output\Win32_MSVC-Headers>kernel32_test path
```

With MinGW

Wine's build system already has support for building tests with a MinGW cross-compiler. See the section above called 'Setup of the MinGW cross-compiling environment' for instructions on how to set things up. When you have a MinGW environment installed all you need to do is rerun `configure` and it should detect the MinGW compiler and tools. Then run 'make crosstest' to start building the tests.

Inside a test

When writing new checks you can either modify an existing test file or add a new one. If your tests are related to the tests performed by an existing file, then add them to that file. Otherwise create a new `.c` file in the tests directory and add that file to the `CTESTS` variable in `Makefile.in`.

A new test file will look something like the following:


```
#include <wine/test.h>
#include <winbase.h>

/* Maybe auxiliary functions and definitions here */

START_TEST(paths)
{
    /* Write your checks there or put them in functions you will call from
     * there
     */
}
```

The test's entry point is the `START_TEST` section. This is where execution will start. You can put all your tests in that section but it may be better to split related checks in functions you will call from the `START_TEST` section. The parameter to `START_TEST` must match the name of the C file. So in the above example the C file would be called `paths.c`.

Tests should start by including the `wine/test.h` header. This header will provide you access to all the testing framework functions. You can then include the windows header you need, but make sure to not include any Unix or Wine specific header: tests must compile on Windows.

You can use `trace` to print informational messages. Note that these messages will only be printed if `'runtest -v'` is being used.

```
trace("testing GlobalAddAtomA");
trace("foo=%d", foo);
```

Then just call functions and use `ok` to make sure that they behaved as expected:

```
ATOM atom = GlobalAddAtomA( "foobar" );
ok( GlobalFindAtomA( "foobar" ) == atom, "could not find atom foobar" );
ok( GlobalFindAtomA( "FOOBAR" ) == atom, "could not find atom FOOBAR" );
```

The first parameter of `ok` is an expression which must evaluate to true if the test was successful. The next parameter is a `printf`-compatible format string which is displayed in case the test failed, and the following optional parameters depend on the format string.

Writing good error messages

The message that is printed when a test fails is *extremely* important.

Someone will take your test, run it on a Windows platform that you don't have access to, and discover that it fails. They will then post an email with the output of the test, and in particular your error message. Someone, maybe you, will then have to figure out from this error message why the test failed.

If the error message contains all the relevant information that will be easy. If not, then it will require modifying the test, finding someone to compile it on Windows, sending the modified version to the original tester and waiting for his reply. In other words, it will be long and painful.

So how do you write a good error message? Let's start with an example of a bad error message:

```
ok(GetThreadPriorityBoost(curthread,&disabled)!=0,
   "GetThreadPriorityBoost Failed");
```

This will yield:

```
thread.c:123: Test failed: GetThreadPriorityBoost Failed
```

Did you notice how the error message provides no information about why the test failed? We already know from the line number exactly which test failed. In fact the error message gives strictly no information that cannot already be obtained by reading the code. In other words it provides no more information than an empty string!

Let's look at how to rewrite it:

```
    BOOL rc;
...
    rc=GetThreadPriorityBoost(curthread,&disabled);
    ok(rc!=0 && disabled==0,"rc=%d error=%ld disabled=%d",
        rc,GetLastError(),disabled);
```

This will yield:

```
thread.c:123: Test failed: rc=0 error=120 disabled=0
```

When receiving such a message, one would check the source, see that it's a call to `GetThreadPriorityBoost`, that the test failed not because the API returned the wrong value, but because it returned an error code. Furthermore we see that `GetLastError()` returned 120 which `winerror.h` defines as `ERROR_CALL_NOT_IMPLEMENTED`. So the source of the problem is obvious: this Windows platform (here Windows 98) does not support this API and thus the test must be modified to detect such a condition and skip the test.

So a good error message should provide all the information which cannot be obtained by reading the source, typically the function return value, error codes, and any function output parameter. Even if more information is needed to fully understand a problem, systematically providing the above is easy and will help cut down the number of iterations required to get to a resolution.

It may also be a good idea to dump items that may be hard to retrieve from the source, like the expected value in a test if it is the result of an earlier computation, or comes from a large array of test values (e.g. index 112 of `_pTestStrA` in `vartest.c`). In that respect, for some tests you may want to define a macro such as the following:

```
#define eq(received, expected, label, type) \
    ok((received) == (expected), "%s: got " type " instead of " type, (label),(received)

...

eq( b, curr_val, "SPI_{GET,SET}BEEP", "%d" );
```

Handling platform issues

Some checks may be written before they pass successfully in Wine. Without some mechanism, such checks would potentially generate hundred of known failures for months each time the tests are being run. This would make it hard to detect new failures caused by a regression. or to detect that a patch fixed a long standing issue.

Thus the Wine testing framework has the concept of platforms and groups of checks can be declared as expected to fail on some of them. In the most common case, one would declare a group of tests as expected to fail in Wine. To do so, use the following construct:

```
todo_wine {
```

```

        SetLastError( 0xdeadbeef );
        ok( GlobalAddAtomA(0) == 0 && GetLastError() == 0xdeadbeef, "failed to add atom 0"
    }

```

On Windows the above check would be performed normally, but on Wine it would be expected to fail, and not cause the failure of the whole test. However. If that check were to succeed in Wine, it would cause the test to fail, thus making it easy to detect when something has changed that fixes a bug. Also note that todo checks are accounted separately from regular checks so that the testing statistics remain meaningful. Finally, note that todo sections can be nested so that if a test only fails on the cygwin and reactos platforms, one would write:

```

todo("cygwin") {
    todo("reactos") {
        ...
    }
}

```

But specific platforms should not be nested inside a todo_wine section since that would be redundant.

When writing tests you will also encounter differences between Windows 9x and Windows NT platforms. Such differences should be treated differently from the platform issues mentioned above. In particular you should remember that the goal of Wine is not to be a clone of any specific Windows version but to run Windows applications on Unix.

So, if an API returns a different error code on Windows 9x and Windows NT, your check should just verify that Wine returns one or the other:

```

ok ( GetLastError() == WIN9X_ERROR || GetLastError() == NT_ERROR, ... );

```

If an API is only present on some Windows platforms, then use LoadLibrary and GetProcAddress to check if it is implemented and invoke it. Remember, tests must run on all Windows platforms. Similarly, conformance tests should not try to correlate the Windows version returned by GetVersion with whether given APIs are implemented or not. Again, the goal of Wine is to run Windows applications (which do not do such checks), and not be a clone of a specific Windows version.

Notes

1. <http://mirzam.it.vu.nl/mingw/>
2. <http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.aspx>

Chapter 5. Internationalization

Adding New Languages

This file documents the necessary procedure for adding a new language to the list of languages that Wine can display system menus and forms in. Adding new translations is not hard as it requires no programming knowledge or special skills.

Language dependent resources reside in files named `somefile_Xx.rc` or `Xx.rc`, where `Xx` is your language abbreviation (look for it in `include/winnls.h`). These are included in a master file named `somefile.rc` or `rsrc.rc`, located in the same directory as the language files.

To add a new language to one of these resources you need to make a copy of the English resource (located in the `somefile_En.rc` file) over to your `somefile_Xx.rc` file, include this file in the master `somefile.rc` file, and edit the new file to translate the English text. You may also need to rearrange some of the controls to better fit the newly translated strings. Test your changes to make sure they properly layout on the screen.

In menus, the character "&" means that the next character will be highlighted and that pressing that letter will select the item. You should place these "&" characters suitably for your language, not just copy the positions from English. In particular, items within one menu should have different highlighted letters.

To get a list of the files that need translating, run the following command in the root of your Wine tree: **find -name "*En.rc"**.

When adding a new language, also make sure the parameters defined in `./dlls/kernel/nls/*.nls` fit your local habits and language.

Chapter 6. Overview

Brief overview of Wine's architecture...

Basic Overview

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

Wine Overview

Wine is often used as a recursive acronym, standing for "Wine Is Not an Emulator". Sometimes it is also known to be used for "Windows Emulator". In a way, both meanings are correct, only seen from different perspectives. The first meaning says that Wine is not a virtual machine, it does not emulate a CPU, and you are not supposed to install Windows nor any Windows device drivers on top of it; rather, Wine is an implementation of the Windows API, and can be used as a library to port Windows applications to Unix. The second meaning, obviously, is that to Windows binaries (.exe files), Wine does look like Windows, and emulates its behaviour and quirks rather closely.

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

Win16 and Win32

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

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

The Wine server

The Wine server is among the most confusing concepts in Wine. What is its function in Wine? Well, to be brief, it provides Inter-Process Communication (IPC), synchronization, and process/thread management. When the wineserver launches, it creates a Unix socket for the current host based on (see below) your home directory's .wine subdirectory (or wherever the WINEPREFIX environment variable points) - all Wine subprocesses launched later connects to the wineserver using this socket. (If a wine-server was not already running, the first Wine process will start up the wineserver in

auto-terminate mode (i.e. the wineserver will then terminate itself once the last Wine process has terminated).)

In earlier versions of Wine the master socket mentioned above was actually created in the configuration directory; either your home directory's `/wine` subdirectory or wherever the `WINEPREFIX` environment variable points to. Since that might not be possible the socket is actually created within the `/tmp` directory with a name that reflects the configuration directory. This means that there can actually be several separate copies of the wineserver running; one per combination of user and configuration directory. Note that you should not have several users using the same configuration directory at the same time; they will have different copies of the wineserver running and this could well lead to problems with the registry information that they are sharing.

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

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

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

Relays, Thunks, and DLL descriptors

Loading a Windows binary into memory isn't that hard by itself, the hard part is all those various DLLs and entry points it imports and expects to be there and function as expected; this is, obviously, what the entire Wine implementation is all about. Wine contains a range of DLL implementations. Each of the implemented (or half-implemented) DLLs (which can be found in the `dlls/` directory) need to make themselves known to the Wine core through a DLL descriptor. These descriptors point to such things as the DLL's resources and the entry point table.

The DLL descriptor and entry point table is generated by the **winebuild** tool (previously just named **build**), taking DLL specification files with the extension `.spec` as input. The output file contains a global constructor that automatically registers the DLL's descriptor with the Wine core at runtime.

Once an application module wants to import a DLL, Wine will look through its list of registered DLLs (if it's not registered, it will look for it on disk). (Failing that, it will look for a real Windows `.DLL` file to use, and look through its imports, etc.) To resolve the module's imports, Wine looks through the entry point table and finds if it's defined there. (If not, it'll emit the error "No handler for ...", which, if the application called the entry point, is a fatal error.)

Since Wine is 32-bit code itself, and if the compiler supports Windows' calling convention, `stdcall` (gcc does), Wine can resolve imports into Win32 code by substituting the addresses of the Wine handlers directly without any thunking layer in between. This eliminates the overhead most people associate with "emulation", and is what the applications expect anyway.

However, if the user specified `WINEDEBUG=+relay`, a thunk layer is inserted between the application imports and the Wine handlers; this layer is known as "relay" because all it does is print out the arguments/return values (by using the argument lists in the DLL descriptor's entry point table), then pass the call on, but it's invaluable for debugging misbehaving calls into Wine code. A similar mechanism also exists between Windows DLLs - Wine can optionally insert thunk layers between them, by using `WINEDEBUG=+snoop`, but since no DLL descriptor information exists for non-Wine DLLs, this is less reliable and may lead to crashes.

For Win16 code, there is no way around thunking - Wine needs to relay between 16-bit and 32-bit code. These thunks switch between the app's 16-bit stack and Wine's 32-bit stack, copies and converts arguments as appropriate, and handles the Win16 mutex. Suffice to say that the kind of intricate stack content juggling this results in, is not exactly suitable study material for beginners.

Core and non-core DLLs

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

Module Overview

KERNEL Module

Needs some content...

GDI Module

X Windows System interface

The X libraries used to implement X clients (such as Wine) do not work properly if multiple threads access the same display concurrently. It is possible to compile the X libraries to perform their own synchronization (initiated by calling `XInitThreads()`). However, Wine does not use this approach. Instead Wine performs its own synchronization using the `wine_tsx11_lock()` / `wine_tsx11_unlock()` functions. This locking protects library access with a critical section, and also arranges things so that X libraries compiled without `-D_REENTRANT` (eg. with global `errno` variable) will work with Wine.

In the past, all calls to X used to go through a wrapper called `TSX. . . ()` (for "Thread Safe X ..."). While it is still being used in the code, it's inefficient as the lock is potentially aquired and released unnecessarily. New code should explicitly aquire the lock.

USER Module

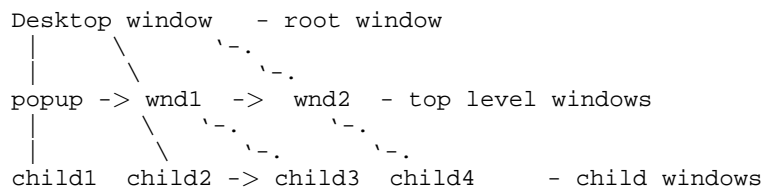
USER implements windowing and messaging subsystems. It also contains code for common controls and for other miscellaneous stuff (rectangles, clipboard, WNet, etc). Wine USER code is located in `windows/`, `controls/`, and `misc/` directories.

Windowing subsystem

`windows/win.c`

`windows/winpos.c`

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



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

Another fairly important concept is *z-order*. It is derived from the ancestor/child hierarchy and is used to determine "above/below" relationship. For instance, in the example above, *z-order* is

`child1->popup->child2->child3->wnd1->child4->wnd2->desktop.`

Current active window ("foreground window" in Win32) is moved to the front of *z-order* unless its top-level ancestor owns popup windows.

All these issues are dealt with (or supposed to be) in `windows/winpos.c` with `SetWindowPos()` being the primary interface to the window manager.

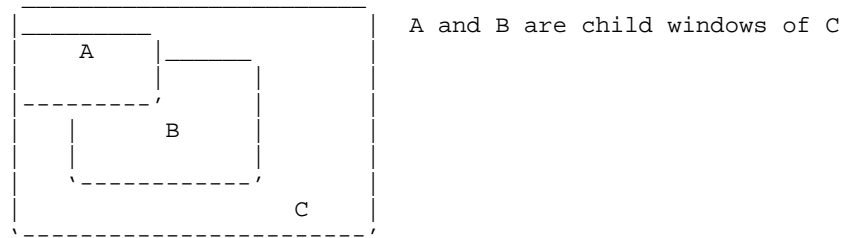
Wine specifics: in default and managed mode each top-level window gets its own X counterpart with desktop window being basically a fake stub. In desktop mode, however, only desktop window has an X window associated with it. Also, `SetWindowPos()` should eventually be implemented via `Begin/End/DeferWindowPos()`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

Visible region, clipping region and update region

`windows/dce.c`

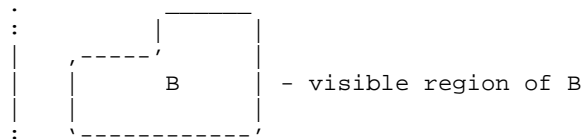
`windows/winpos.c`

windows/painting.c



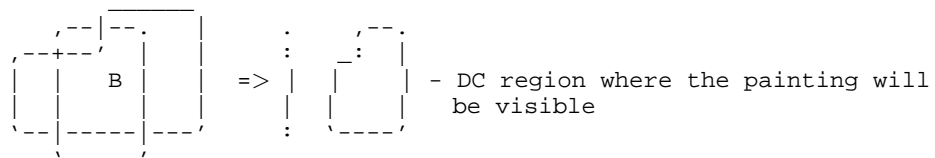
Visible region determines which part of the window is not obscured by other windows. If a window has the `WS_CLIPCHILDREN` style then all areas below its children are considered invisible. Similarly, if the `WS_CLIPSIBLINGS` bit is in effect then all areas obscured by its siblings are invisible. Child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

B has a `WS_CLIPSIBLINGS` style:



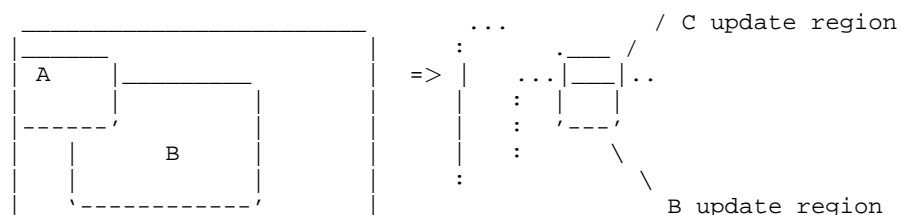
When the program requests a *display context* (DC) for a window it can specify an optional clipping region that further restricts the area where the graphics output can appear. This area is calculated as an intersection of the visible region and a clipping region.

Program asked for a DC with a clipping region:



When the window manager detects that some part of the window became visible it adds this area to the update region of this window and then generates `WM_ERASEBKGD` and `WM_PAINT` messages. In addition, `WM_NCPAINT` message is sent when the uncovered area intersects a nonclient part of the window. Application must reply to the `WM_PAINT` message by calling the `BeginPaint()/EndPaint()` pair of functions. `BeginPaint()` returns a DC that uses accumulated update region as a clipping region. This operation cleans up invalidated area and the window will not receive another `WM_PAINT` until the window manager creates a new update region.

A was moved to the left:





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

Messaging subsystem

`windows/queue.c`

`windows/message.c`

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

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

Message priority:

First the system looks for sent messages, then for posted messages, then for hardware messages, then it checks if the queue has the "dirty window" bit set, and, finally, it checks for expired timers. See `windows/message.c`.

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

There are several tricky moments (open for discussion) -

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

by David Charlap

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

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

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

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

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

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

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

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

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

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

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

There are a few solutions to this problem.

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

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

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

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

As for the pros and cons:

The X-windows solution is probably the easiest. The problem is that users generally don't like having to wait for the focus to change before they start typing. On many occasions, you can type and the characters end up in the wrong window because some-

thing (usually heavy system load) is preventing the focus change from happening in a timely manner.

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

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

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

Wine/Windows DLLs

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

Pros of Native DLLs

Native DLLs of course guarantee 100% compatibility for routines they implement. For example, using the native USER DLL would maintain a virtually perfect and Windows 95-like look for window borders, dialog controls, and so on. Using the built-in Wine version of this library, on the other hand, would produce a display that does not precisely mimic that of Windows 95. Such subtle differences can be engendered in other important DLLs, such as the common controls library `COMMCTRL` or the common dialogs library `COMMDLG`, when built-in Wine DLLs outrank other types in load order.

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

Cons of Native DLLs

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

Finally, occasionally built-in Wine DLLs implement more features than the corresponding native Windows DLLs. Probably the most important example of such behavior is the integration of Wine with X provided by Wine's built-in USER DLL. Should the native Windows USER library take load-order precedence, such features

as the ability to use the clipboard or drag-and-drop between Wine windows and X windows will be lost.

Deciding Between Native and Built-In DLLs

Clearly, there is no one rule-of-thumb regarding which load-order to use. So, you must become familiar with what specific DLLs do and which other DLLs or features a given library interacts with, and use this information to make a case-by-case decision.

Load Order for DLLs

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

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

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

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

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

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

Understanding What DLLs Do

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

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

ADVAPI32.DLL:	32-bit application advanced programming interfaces like crypto, systeminfo, security and event logging
AVIFILE.DLL:	32-bit application programming interfaces for the Audio Video Interleave (AVI) Windows-specific Microsoft audio-video standard
COMMCTRL.DLL:	16-bit common controls
COMCTL32.DLL:	32-bit common controls
COMDLG32.DLL:	32-bit common dialogs
COMMDBG.DLL:	16-bit common dialogs
COMPOBJ.DLL:	OLE 16- and 32-bit compatibility libraries
CRTDLL.DLL:	Microsoft C runtime
DCIMAN.DLL:	16-bit
DCIMAN32.DLL:	32-bit display controls
DDEML.DLL:	DDE messaging
D3D*.DLL	DirectX/Direct3D drawing libraries

```

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

```


Chapter 7. Debug Logging

To better manage the large volume of debugging messages that Wine can generate, we divide the messages on a component basis, and classify them based on the severity of the reported problem. Therefore a message belongs to a *channel* and a *class* respectively.

This section will describe the debugging classes, how you can create a new debugging channel, what the debugging API is, and how you can control the debugging output. A picture is worth a thousand words, so here are a few examples of the debugging API in action:

```
ERR("lock_count == 0 ... please report\n");
FIXME("Unsupported RTL style!\n");
WARN(": file seems to be truncated!\n");
TRACE("[%p]: new horz extent = %d\n", hwnd, extent );
MESSAGE( "Could not create graphics driver '%s'\n", buffer );
```

Debugging classes

A debugging class categorizes a message based on the severity of the reported problem. There is a fixed set of classes, and you must carefully choose the appropriate one for your messages. There are five classes of messages:

FIXME

Messages in this class are meant to signal unimplemented features, known bugs, etc. They serve as a constant and active reminder of what needs to be done.

ERR

Messages in this class indicate serious errors in Wine, such as as conditions that should never happen by design.

WARN

These are warning messages. You should report a warning when something unwanted happens, and the function can not deal with the condition. This is seldomly used since proper functions can usually report failures back to the caller. Think twice before making the message a warning.

TRACE

These are detailed debugging messages that are mainly useful to debug a component. These are turned off unless explicitly enabled.

MESSAGE

There messages are intended for the end user. They do not belong to any channel. As with warnings, you will seldomly need to output such messages.

Debugging channels

Each component is assigned a debugging channel. The identifier of the channel must be a valid C identifier (reserved word like `int` or `static` are premitted). To use a new channel, simply use it in your code. It will be picked up automatically by the build process.

Typically, a file contains code pertaining to only one component, and as such, there is only one channel to output to. You can declare a default channel for the file using the `WINE_DEFAULT_DEBUG_CHANNEL()` macro:

```
#include "wine/debug.h"

WINE_DEFAULT_DEBUG_CHANNEL(xxx);
...

    FIXME("some unimplemented feature", ...);
...
    if (zero != 0)
        ERR("This should never be non-null: %d", zero);
...
```

In rare situations there is a need to output to more than one debug channel per file. In such cases, you need to declare all the additional channels at the top of the file, and use the `_`-version of the debugging macros:

```
#include "wine/debug.h"

WINE_DEFAULT_DEBUG_CHANNEL(xxx);
WINE_DECLARE_DEBUG_CHANNEL(yyy);
WINE_DECLARE_DEBUG_CHANNEL(zzz);
...

    FIXME("this one goes to xxx channel");
...
    FIXME_(yyy)("Some other msg for the yyy channel");
...
    WARN_(zzz)("And yet another msg on another channel!");
...
```

Are we debugging?

To test whether the debugging channel `xxx` is enabled, use the `TRACE_ON`, `WARN_ON`, `FIXME_ON`, or `ERR_ON` macros. For example:

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

You should normally need to test only if `TRACE_ON`, all the others are very seldomly used. With careful coding, you can avoid the use of these macros, which is generally desired.

Helper functions

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

The function is defined in `wine/debug.h` and has the following prototype:

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

It takes a pointer to the resource id and returns a nicely formatted string of the identifier (which can be a string or a number, depending on the value of the high word). Numbers are formatted as such:

```
#xxxx
```

while strings as:

```
'some-string'
```

Simply use it in your code like this:

```
#include "wine/debug.h"

...

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

Many times strings need to be massaged before output: they may be NULL, contain control characters, or they may be too long. Similarly, Unicode strings need to be converted to ASCII for usage with the debugging API. For all this, you can use the `debugstr_[aw]n?` family of functions:

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

Controlling the debugging output

It is possible to turn on and off debugging output from within the debugger using the `set` command. Please see the WineDbg Command Reference section for how to do this.

Another way to conditionally log debug output (e.g. in case of very large installers which may create gigabytes of log output) is to create a pipe:

```
$ mknod /tmp/debug_pipe p
```

and then to run wine like that:

```
$ WINEDEBUG=+relay,+snoop wine setup.exe &>/tmp/debug_pipe
```

Since the pipe is initially blocking (and thus wine as a whole), you have to activate it by doing:

```
$ cat /tmp/debug_pipe
```

(press Ctrl-C to stop pasting the pipe content)

Once you are about to approach the problematic part of the program, you just do:

```
$ cat /tmp/debug_pipe >/tmp/wine.log
```

to capture specifically the part that interests you from the pipe without wasting excessive amounts of HDD space and slowing down installation considerably.

The *WINEDEBUG* environment variable controls the output of the debug messages. It has the following syntax: *WINEDEBUG= [yyy]#xxx[, [yyy1]#xxx1]**

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

```
WINEDEBUG=+reg,-file
```

enables all messages on the *reg* channel and disables all messages on the *file* channel.

- when the optional class argument (yyy) is present, then the statement will enable (+)/disable(-) messages for the given channel (xxx) only on the given class. For example:

```
WINEDEBUG=trace+reg,warn-file
```

enables trace messages on the *reg* channel and disables warning messages on the *file* channel.

- also, the pseudo-channel *all* is also supported and it has the intuitive semantics:

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

So, for example:

```
WINEDEBUG=warn-all  -- disables all warning messages.
```

Also, note that at the moment:

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

Compiling Out Debugging Messages

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

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

This will result in an executable that, when stripped, is about 15%-20% smaller. Note, however, that you will not be able to effectively debug Wine without these messages.

This feature has not been extensively tested--it may subtly break some things.

A Few Notes on Style

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

```
yyy:xxx:fff <message>
```

where:

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

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

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

```
TRACE("(%d, %p, ...)\n", par1, par2, ...);
```

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

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

- try to output one line per message. That is, the format string should contain only one `\n` and it should always appear at the end of the string. (there are many reasons for this requirement, one of them is that each debug macro adds things to the beginning of the line)
- if you want to name a parameter, use `= :`

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


Chapter 8. COM/OLE in Wine

Writing OLE Components for Wine

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

Macros to define a COM interface

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

It is based on the following assumptions:

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

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

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

Let's take Direct3D as an example:

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

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

```
#endif
```

Comments:

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

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

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

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

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

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

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

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

Bindings in C

In C this gives:

```
typedef struct IDirect3DVtbl IDirect3DVtbl;
struct IDirect3D {
    IDirect3DVtbl* lpVtbl;
};
struct IDirect3DVtbl {
    HRESULT (*fnQueryInterface)(IDirect3D* me, REFIID riid, LPVOID* ppvObj);
    ULONG (*fnAddRef)(IDirect3D* me);
    ULONG (*fnRelease)(IDirect3D* me);
    HRESULT (*fnInitialize)(IDirect3D* me, REFIID a);
```



```

        HRESULT (*fnEnumDevices)(IDirect3D* me, LPD3DENUMDEVICESCALLBACK a, LPVOID b);
        HRESULT (*fnCreateLight)(IDirect3D* me, LPDIRECT3DLIGHT* a, IUnknown* b);
        HRESULT (*fnCreateMaterial)(IDirect3D* me, LPDIRECT3DMATERIAL* a, IUnknown* b);
        HRESULT (*fnCreateViewport)(IDirect3D* me, LPDIRECT3DVIEWPORT* a, IUnknown* b);
        HRESULT (*fnFindDevice)(IDirect3D* me, LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESULT b);
};

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

```

Comments:

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

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

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

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

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

Bindings in C++

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

```

typedef struct IDirect3D: public IUnknown {
    private: HRESULT (*fnInitialize)(IDirect3D* me, REFIID a);
    public: inline HRESULT Initialize(REFIID a) { return ((IDirect3D*)t.lpVtbl)->fnInit
    private: HRESULT (*fnEnumDevices)(IDirect3D* me, LPD3DENUMDEVICESCALLBACK a, LPVOID b)
    public: inline HRESULT EnumDevices(LPD3DENUMDEVICESCALLBACK a, LPVOID b)
        { return ((IDirect3D*)t.lpVtbl)->fnEnumDevices(this,a,b); };
    private: HRESULT (*fnCreateLight)(IDirect3D* me, LPDIRECT3DLIGHT* a, IUnknown* b);
    public: inline HRESULT CreateLight(LPDIRECT3DLIGHT* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateLight(this,a,b); };
    private: HRESULT (*fnCreateMaterial)(IDirect3D* me, LPDIRECT3DMATERIAL* a, IUnknown* b);
    public: inline HRESULT CreateMaterial(LPDIRECT3DMATERIAL* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateMaterial(this,a,b); };
    private: HRESULT (*fnCreateViewport)(IDirect3D* me, LPDIRECT3DVIEWPORT* a, IUnknown* b);
    public: inline HRESULT CreateViewport(LPDIRECT3DVIEWPORT* a, IUnknown* b)
        { return ((IDirect3D*)t.lpVtbl)->fnCreateViewport(this,a,b); };
    private: HRESULT (*fnFindDevice)(IDirect3D* me, LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESULT b);
    public: inline HRESULT FindDevice(LPD3DFINDDEVICESEARCH a, LPD3DFINDDEVICERESULT b)

```

```

        { return ((IDirect3D*)t.lpVtbl)->fnFindDevice(this,a,b); };
};

```

Comments:

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

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

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

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

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

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

Implementing a COM interface.

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

```

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

static ICOM_VTABLE(IDirect3D) d3dvt;

// implement the IDirect3D methods here

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

// ...

static ICOM_VTABLE(IDirect3D) d3dvt = {
    ICOM_MSVTABLE_COMPAT_DummyRTTIVALUE
    IDirect3D_fnQueryInterface,
    IDirect3D_fnAdd,

```

```

        IDirect3D_fnAdd2,
        IDirect3D_fnInitialize,
        IDirect3D_fnSetWidth
    };

```

Comments:

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

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

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

Finally we initialize the virtual table.

Chapter 9. Wine and OpenGL

What is needed to have OpenGL support in Wine

Basically, if you have a Linux OpenGL ABI compliant libGL (<http://oss.sgi.com/projects/ogl-sample/ABI/>¹) installed on your computer, you should have everything that is needed.

To be more clear, I will detail one step after another what the **configure** script checks.

If, after Wine compiles, OpenGL support is not compiled in, you can always check `config.log` to see which of the following points failed.

Header files

The needed header files to build OpenGL support in Wine are :

<code>gl.h:</code>	the definition of all OpenGL core functions, types and enumerants
<code>glx.h:</code>	how OpenGL integrates in the X Window environment
<code>glext.h:</code>	the list of all registered OpenGL extensions

The latter file (`glext.h`) is, as of now, not necessary to build Wine. But as this file can be easily obtained from SGI (<http://oss.sgi.com/projects/ogl-sample/ABI/glext.h>²), and that all OpenGL should provide one, I decided to keep it here.

OpenGL library itself

To check for the presence of 'libGL' on the system, the script checks if it defines the `glXCreateContext` function.

glXGetProcAddressARB function

The core of Wine's OpenGL implementation (at least for all extensions) is the `glXGetProcAddressARB` function. Your OpenGL library needs to have this function defined for Wine to be able to support OpenGL.

How it all works

The core OpenGL function calls are the same between Windows and Linux. So what is the difficulty to support it in Wine ? Well, there are two different problems :

1. the interface to the windowing system is different for each OS. It's called 'GLX' for Linux (well, for X Window) and 'wgl' for Windows. Thus, one need first to emulate one (wgl) with the other (GLX).
2. the calling convention between Windows (the 'Pascal' convention or 'stdcall') is different from the one used on Linux (the 'C' convention or 'cdecl'). This means that each call to an OpenGL function must be 'translated' and cannot be used directly by the Windows program.

Add to this some brain-dead programs (using GL calls without setting-up a context or deleting three time the same context) and you have still some work to do :-)

The Windowing system integration

This integration is done at two levels :

1. At GDI level for all pixel format selection routines (ie choosing if one wants a depth / alpha buffer, the size of these buffers, ...) and to do the 'page flipping' in double buffer mode. This is implemented in `dlls/x11drv/opengl.c` (all these functions are part of Wine's graphic driver function pointer table and thus could be reimplemented if ever Wine works on another Windowing system than X).
2. In the `OpenGL32.DLL` itself for all other functionalities (context creation / deletion, querying of extension functions, ...). This is done in `dlls/opengl32/wgl.c`.

The thunks

The thunks are the Wine code that does the calling convention translation and they are auto-generated by a Perl script. In Wine's CVS tree, these thunks are already generated for you. Now, if you want to do it yourself, there is how it all works....

The script is located in `dlls/opengl32` and is called **make_opengl**. It requires Perl5 to work and takes two arguments :

1. The first is the path to the OpenGL registry. Now, you will all ask 'but what is the OpenGL registry ?' :-). Well, it's part of the OpenGL sample implementation source tree from SGI (more informations at this URL : <http://oss.sgi.com/projects/ogl-sample/>³).

To summarize, these files contain human-readable but easily parsed information on ALL OpenGL core functions and ALL registered extensions (for example the prototype, the OpenGL version, ...).

2. the second is the OpenGL version to 'simulate'. This fixes the list of functions that the Windows application can link directly to without having to query them from the OpenGL driver. Windows is based, for now, on OpenGL 1.1, but the thunks that are in the CVS tree are generated for OpenGL 1.2.

This option can have three values: 1.0, 1.1 and 1.2.

This script generates three files :

1. `opengl32.spec` gives Wine's linker the signature of all function in the `OpenGL32.DLL` library so that the application can link them. Only 'core' functions are listed here.
2. `opengl_norm.c` contains all the thunks for the 'core' functions. Your OpenGL library must provide ALL the function used in this file as these are not queried at run time.
3. `opengl_ext.c` contains all the functions that are not part of the 'core' functions. Contrary to the thunks in `opengl_norm.c`, these functions do not depend at all on what your libGL provides.

In fact, before using one of these thunks, the Windows program first needs to 'query' the function pointer. At this point, the corresponding thunk is useless.

But as we first query the same function in libGL and store the returned function pointer in the thunk, the latter becomes functional.

Known problems

When running an OpenGL application, the screen flickers

Due to restrictions (that do not exist in Windows) on OpenGL contexts, if you want to prevent the screen to flicker when using OpenGL applications (all games are using double-buffered contexts), you need to set the following option in your `~/.wine/config` file in the `[x11drv]` section:

```
DesktopDoubleBuffered = Y
```

and to run Wine in desktop mode.

Unknown extension error message:

```
Extension defined in the OpenGL library but NOT in opengl_ext.c...
Please report (lionel.ulmer@free.fr) !
```

This means that the extension requested by the application is found in the libGL used by Linux (ie the call to `glXGetProcAddressARB` returns a non-NULL pointer) but that this string was NOT found in Wine's extension registry.

This can come from two causes:

1. The `opengl_ext.c` file is too old and needs to be generated again.
2. Use of obsolete extensions that are not supported anymore by SGI or of 'private' extensions that are not registered. An example of the former are `glMTexCoord2fSGIS` and `glSelectTextureSGIS` as used by Quake 2 (and apparently also by old versions of Half Life). If documentation can be found on these functions, they can be added to Wine's extension set.

If you have this, run with `WINEDEBUG=+opengl` and send me `<lionel.ulmer@free.fr>` the TRACE.

libopengl32.so is built but it is still not working

This may be caused by some missing functions required by `opengl_norm.c` but that your Linux OpenGL library does not provide.

To check for this, do the following steps :

1. create a dummy `.c` file :

```
int main(void)
{
    return 0;
}
```

2. try to compile it by linking both libwine and libopengl32 (this command line supposes that you installed the Wine libraries in /usr/local/lib, YMMV) :

```
gcc dummy.c -L/usr/local/lib -lwine -lopengl32
```
3. if it works, the problem is somewhere else (and you can send me an email). If not, you could re-generate the thunk files for OpenGL 1.1 for example (and send me your OpenGL version so that this problem can be detected at configure time).

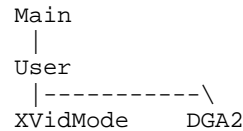
Notes

1. <http://oss.sgi.com/projects/ogl-sample/ABI/>
2. <http://oss.sgi.com/projects/ogl-sample/ABI/glexth.h>
3. <http://oss.sgi.com/projects/ogl-sample/>

Chapter 10. Outline of DirectDraw Architecture

This is an outline of the architecture. Many details are skipped, but hopefully this is useful.

DirectDraw inheritance tree



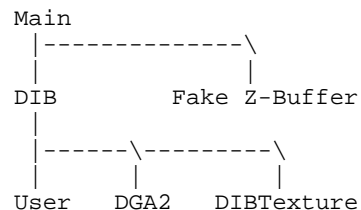
Most of the DirectDraw functionality is implemented in a common base class. Derived classes are responsible for providing display mode functions (Enum, Set, Restore), GetCaps, GetDevice identifier and internal functions called to create primary and backbuffer surfaces.

User provides for DirectDraw capabilities based on drawing to a Wine window. It uses the User DirectDrawSurface implementation for primary and backbuffer surfaces.

XVidMode attempt to use the XFree86 VidMode extension to set the display resolution to match the parameters to SetDisplayMode.

DGA2 attempt to use the XFree86 DGA 2.x extension to set the display resolution and direct access to the framebuffer, if the full-screen-exclusive cooperative level is used. If not, it just uses the User implementation.

DirectDrawSurface inheritance tree



Main provides a very simple base class that does not implement any of the image-related functions. Therefore it does not place any constraints on how the surface data is stored.

DIB stores the surface data in a DIB section. It is used by the Main DirectDraw driver to create off-screen surfaces.

User implements primary and backbuffer surfaces for the User DirectDraw driver. If it is a primary surface, it will attempt to keep itself synchronized to the window.

DGA2 surfaces claims an appropriate section of framebuffer space and lets DIB build its DIB section on top of it.

Fake Z-Buffer surfaces are used by Direct3D to indicate that a primary surface has an associated z-buffer. For a first implementation, it doesn't need to store any image data since it is just a placeholder.

(Actually 3D programs will rarely use Lock or GetDC on primary surfaces, backbuffers or z-buffers so we may want to arrange for lazy allocation of the DIB sections.)

Interface Thunks

Only the most recent version of an interface needs to be implemented. Other versions are handled by having thunks convert their parameters and call the root version.

Not all interface versions have thunks. Some versions could be combined because their parameters were compatible. For example if a structure changes but the structure has a `dwSize` field, methods using that structure are compatible, as long as the implementation remembers to take the `dwSize` into account.

Interface thunks for Direct3D are more complicated since the paradigm changed between versions.

Logical Object Layout

The objects are split into the generic part (essentially the fields for `Main`) and a private part. This is necessary because some objects can be created with `CoCreateInstance`, then `Initialized` later. Only at initialization time do we know which class to use. Each class except `Main` declares a `Part` structure and adds that to its `Impl`.

For example, the `DIBTexture DirectDrawSurface` implementation looks like this:

```
struct DIBTexture_DirectDrawSurfaceImpl_Part
{
    union DIBTexture_data data; /*declared in the real header*/
};

typedef struct
{
    struct DIB_DirectDrawSurfaceImpl_Part dib;
    struct DIBTexture_DirectDrawSurfaceImpl_Part dibtexture;
} DIBTexture_DirectDrawSurfaceImpl;
```

So the `DIBTexture` surface class is derived from the `DIB` surface class and it adds one piece of data, a union.

`Main` does not have a `Part` structure. Its fields are stored in `IDirectDrawImpl/IDirectDrawSurfaceImpl`.

To access private data, one says

```
DIBTexture_DirectDrawSurfaceImpl* priv = This->private;
do_something_with(priv->dibtexture.data);
```

Creating Objects

Classes have two functions relevant to object creation, `Create` and `Construct`. To create a new object, the class' `Create` function is called. It allocates enough memory for `IDirectDrawImpl` or `IDirectDrawSurfaceImpl` as well as the private data for derived classes and then calls `Construct`.

Each class's `Construct` function calls the base class's `Construct`, then does the necessary initialization.

For example, creating a primary surface with the user `ddraw` driver calls `User_DirectDrawSurface_Create` which allocates memory for the object and calls `User_DirectDrawSurface_Construct` to initialize it. This calls `DIB_DirectDrawSurface_Construct` which calls `Main_DirectDrawSurface_Construct`.

Chapter 11. Wine and Multimedia

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

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

Overview

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

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

There are two specific low level drivers (`msacm.driv` for wave input/output, `midimap.driv` for MIDI output only), whose role is:

- help choosing one low level driver between many
- add the possibility to convert streams (ie ADPCM => PCM) (this is useful if the format required by the application for playback isn't supported by the soundcard).
- add the possibility to filter a stream (adding echo, equalizer... to a wave stream), or modify the instruments that have to be played (MIDI).

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

Low level layers

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

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

(Wave form) Audio

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

OSS implementation

The low level audio driver is currently only implemented for the `OpenSoundSystem` (OSS) as supplied in the Linux and FreeBSD kernels by 4Front Technologies¹. The presence of this driver is checked by `configure` (depends on the `<sys/soundcard.h>` file). Source code resides in `dlls/winmm/wineoss/audio.c`.

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

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

TODO:

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

Other sub systems

Support is also provided for ALSA, aRts, NAS, Jack, AudioIO, but with less intensive debugging than the OSS.

Esound isn't supported yet. ALSA support still needs a couple of refinements.

MIDI

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

OSS driver

The low level audio driver is currently only implemented for the OpenSoundSystem (OSS) as supplied in the Linux and FreeBSD kernels by 4Front Technologies². The presence of this driver is checked by configure (depends on the <sys/soundcard.h> file, and also some specific defines because MIDI is not supported on all OSes by OSS). Source code resides in dlls/winmm/wineoss/midi.c

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

TODO:

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

Other sub systems

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

Could also implement a software synthesizer, either inside Wine or using using MIDI loop back devices in an external program (like timidity). The only trouble is that timidity is GPL'ed... Note: this could be achieved using the ALSA sequencer and Timidity being used as a server.

Mixer

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

OSS implementation

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

TODO:

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

Other sub systems

TODO:

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

Aux

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

OSS driver

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

TODO:

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

Wine OSS

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

Joystick

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

TODO:

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

Wave mapper (msacm.driv)

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

Built-in

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

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

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

TODO:

- check for correctness and robustness

Native

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

MIDI mapper

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

Built-in

A built-in MIDI mapper can be found in dlls/winmm/midimap/. It partly provides the same functionality as the Windows' one. It allows to pick up destination channels

(you can map a given channel to a specific playback device channel (see the configuration bits for more details).

TODO:

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

Native

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

TODO:

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

Mid level drivers (MCI)

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

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

- use `MMSYSTEM` multitasking caps instead of the home grown

CDAUDIO

Built-in

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

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

TODO:

- add support for other cdaudio drivers (Solaris...)
- add support for multiple cdaudio devices (plus a decent configuration scheme)

Native

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

MCIWAVE

Built-in

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

FIXME:

- The `MCI_STATUS` command is broken.

TODO:

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

Native

Native MCIWAVE works also correctly.

MCISEQ (MIDI sequencer)

Built-in

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

TODO:

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

Native

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

FIXME:

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

MCIANIM

Built-in

The implementation is in `dlls/winmm/mcianim/`.

TODO:

- implement it, probably using xanim or something similar.

Native

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

MCIavi

Built-in

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

TODO:

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

Native

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

High level layers

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

TODO:

- it seems that some programs check what's installed in registry against the value returned by drivers. Wine is currently broken regarding this point.
- check thread-safeness for MMSYSTEM and WINMM entry points
- unicode entry points are badly supported

MCI skeleton

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

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

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

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

TODO:

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

MCI multi-tasking

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

TODO:

- `mmTaskXXX` functions are currently broken because the 16 loader does not support binary command lines => provide Wine's own `mmtask.tsk` not using binary command line.

Timers

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

TODO:

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

MMIO

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

sndPlayXXX functions

Seem to work correctly.

Multimedia configuration

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

Drivers

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

MCI

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

```
[mci]
cdaudio=mcicda.drv
sequencer=mciseq.drv
waveaudio=mcivave.drv
avivideo=mcivavi.drv
videodisc=mcipionr.drv
vcr=mcivisca.drv
MPEGVideo=mcigtz.drv
```

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

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

TODO:

- use a default registry setting to bypass this (ugly) configuration model
- we need also a generic tool to let the end user pick up his/her driver depending on the hardware present on the machine. model

Low level drivers

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

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

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

Midi mapper

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

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

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

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\MediaProperties\PrivateProperties\M
```

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

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

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\MediaProperties\PrivateProperties\M
```

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

ACM

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

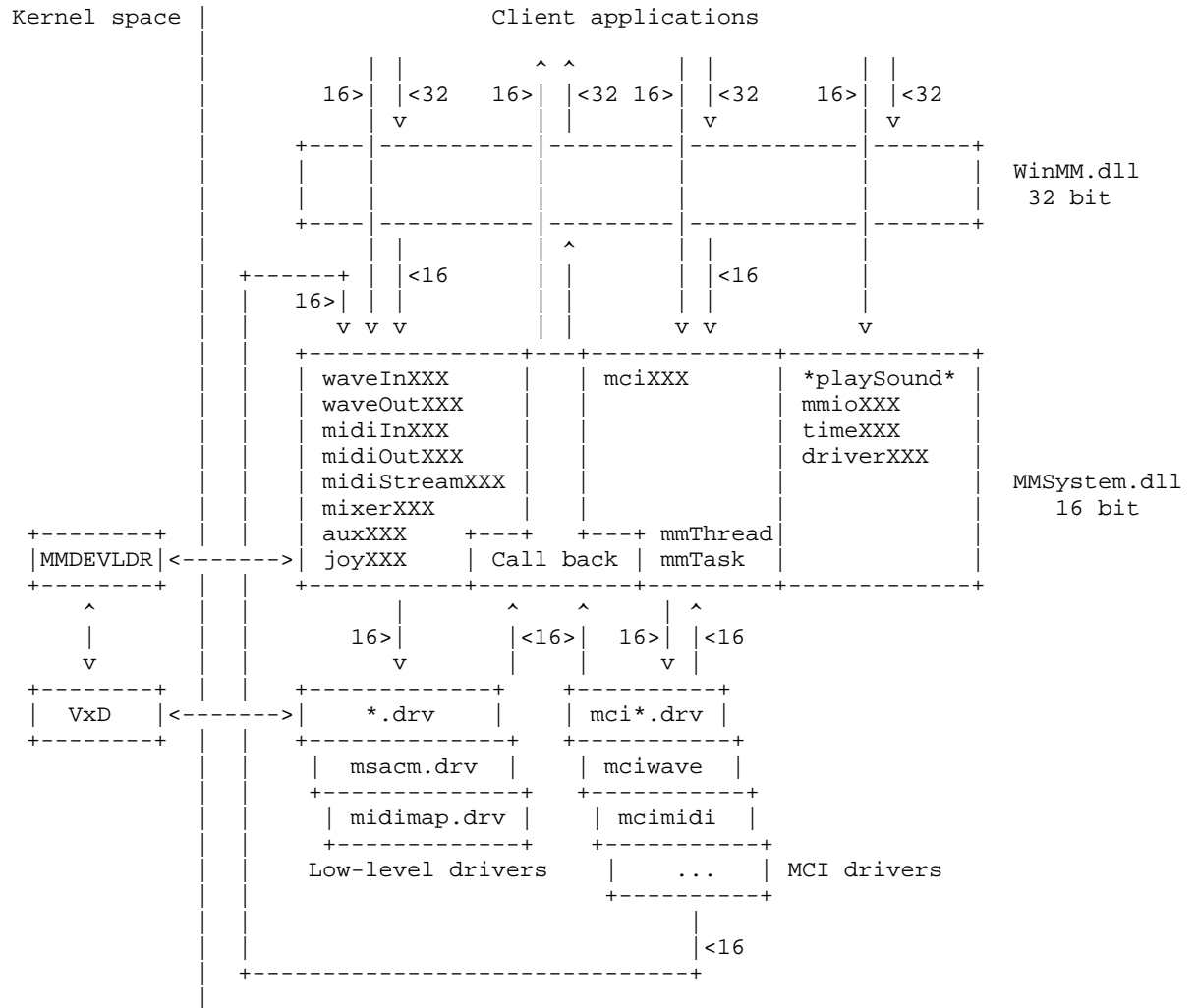
VIDC

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

Multimedia architecture

Windows 9x multimedia architecture

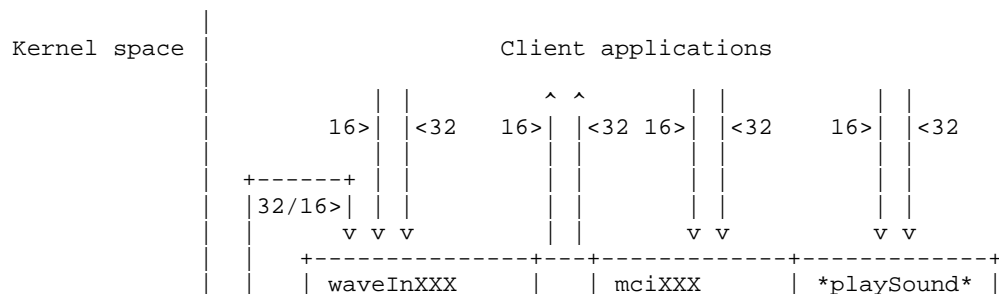
|

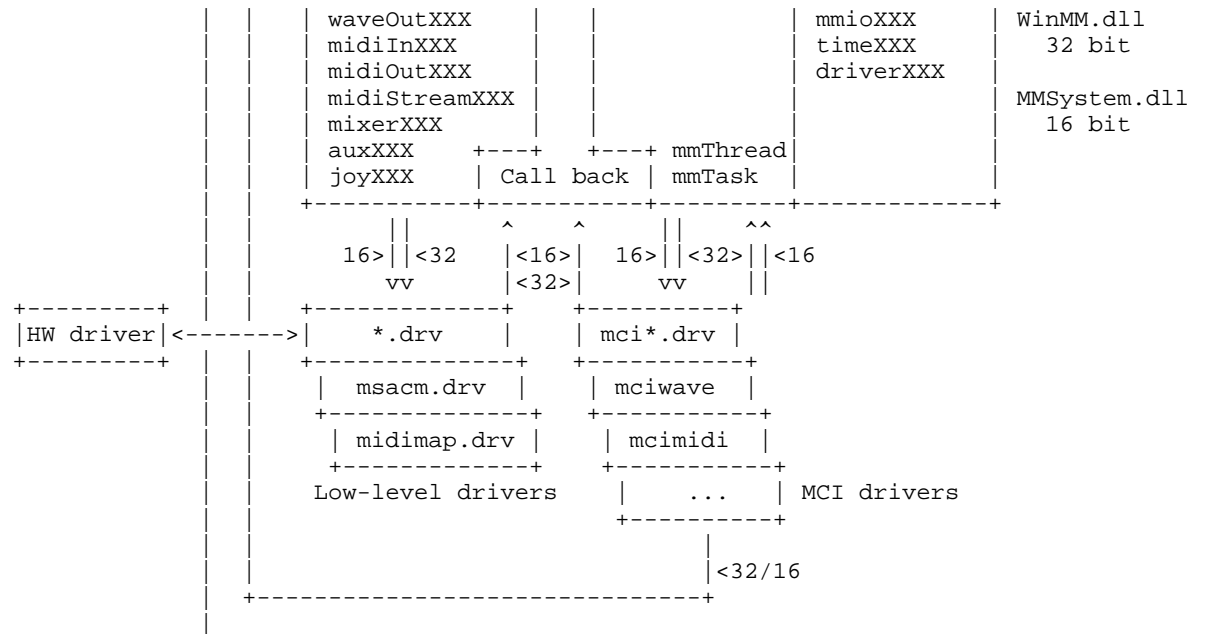


The important points to notice are:

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

Wine multimedia architecture





From the previous drawings, the most noticeable differences are:

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

MS ACM DIIs

Contents

tbd

Status

tbd

Caching

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

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

, are kept for values:

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

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

Notes

1. <http://www.4front-tech.com/>
2. <http://www.4front-tech.com/>

Chapter 12. Low-level Implementation

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

Keyboard

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

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

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

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

```
static const int main_key_scan[MAIN_LEN] =
{
/* this is my (102-key) keyboard layout, sorry if it doesn't quite match yours */
0x29,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,
0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,
0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x2B,
0x2C,0x2D,0x2E,0x2F,0x30,0x31,0x32,0x33,0x34,0x35,
0x56 /* the 102nd key (actually to the right of 1-shift) */
};
```

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

However, for most international 102-key keyboards, we have done it easy for you. The scancode layout for these already pretty much matches the physical layout in the `main_key_scan`, so all you need to do is to go through all the keys that generate characters on your main keyboard (except spacebar), and stuff those into an appropriate table. The only exception is that the 102nd key, which is usually to the left of the first key of the last line (usually **Z**), must be placed on a separate line after the last line.

For example, my Norwegian keyboard looks like this

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

Tab Q W E R T Y U I O P Å ^
    ..~
    Enter
Caps A S D F G H J K L Ø Æ *
Lock

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

Ctrl  Alt      Spacebar      AltGr  Ctrl
```

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

This keyboard is defined as follows:

```
static const char main_key_NO[MAIN_LEN][4] =
```

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

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

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

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

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

```
WINEDEBUG+=key,+keyboard wine > key.log 2>&1
```

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

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

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

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

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

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

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

and submit it as explained above.

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

Undocumented APIs

Some background: On the i386 class of machines, stack entries are usually `dword` (4 bytes) in size, little-endian. The stack grows downward in memory. The stack pointer, maintained in the `esp` register, points to the last valid entry; thus, the operation of

pushing a value onto the stack involves decrementing `esp` and then moving the value into the memory pointed to by `esp` (i.e., `push p` in assembly resembles `*(--esp) = p;` in C). Removing (popping) values off the stack is the reverse (i.e., `pop p` corresponds to `p = *(esp++);` in C).

In the `stdcall` calling convention, arguments are pushed onto the stack right-to-left. For example, the C call `myfunction(40, 20, 70, 30);` is expressed in Intel assembly as:

```
push 30
push 70
push 20
push 40
call myfunction
```

The called function is responsible for removing the arguments off the stack. Thus, before the call to `myfunction`, the stack would look like:

```

[local variable or temporary]
[local variable or temporary]
30
70
20
esp -> 40
```

After the call returns, it should look like:

```

[local variable or temporary]
esp -> [local variable or temporary]
```

To restore the stack to this state, the called function must know how many arguments to remove (which is the number of arguments it takes). This is a problem if the function is undocumented.

One way to attempt to document the number of arguments each function takes is to create a wrapper around that function that detects the stack offset. Essentially, each wrapper assumes that the function will take a large number of arguments. The wrapper copies each of these arguments into its stack, calls the actual function, and then calculates the number of arguments by checking `esp` before and after the call.

The main problem with this scheme is that the function must actually be called from another program. Many of these functions are seldom used. An attempt was made to aggressively query each function in a given library (`ntdll.dll`) by passing 64 arguments, all 0, to each function. Unfortunately, Windows NT quickly goes to a blue screen of death, even if the program is run from a non-administrator account.

Another method that has been much more successful is to attempt to figure out how many arguments each function is removing from the stack. This instruction, `ret hhll` (where `hhll` is the number of bytes to remove, i.e. the number of arguments times 4), contains the bytes `0xc2 11 hh` in memory. It is a reasonable assumption that few, if any, functions take more than 16 arguments; therefore, simply searching for `hh == 0 && 11 < 0x40` starting from the address of a function yields the correct number of arguments most of the time.

Of course, this is not without errors. `ret 0011` is not the only instruction that can have the byte sequence `0xc2 11 0x0`; for example, `push 0x000040c2` has the byte sequence `0x68 0xc2 0x40 0x0 0x0`, which matches the above. Properly, the utility should look for this sequence only on an instruction boundary; unfortunately, finding instruction boundaries on an i386 requires implementing a full disassembler -- quite

a daunting task. Besides, the probability of having such a byte sequence that is not the actual return instruction is fairly low.

Much more troublesome is the non-linear flow of a function. For example, consider the following two functions:

```
somefunction1:
    jmp  somefunction1_impl

somefunction2:
    ret  0004

somefunction1_impl:
    ret  0008
```

In this case, we would incorrectly detect both `somefunction1` and `somefunction2` as taking only a single argument, whereas `somefunction1` really takes two arguments.

With these limitations in mind, it is possible to implement more stubs in Wine and, eventually, the functions themselves.

Accelerators

There are *three* differently sized accelerator structures exposed to the user:

1. Accelerators in NE resources. This is also the internal layout of the global handle HACCEL (16 and 32) in Windows 95 and Wine. Exposed to the user as Win16 global handles HACCEL16 and HACCEL32 by the Win16/Win32 API. These are 5 bytes long, with no padding:

```
BYTE    fVirt;
WORD    key;
WORD    cmd;
```

2. Accelerators in PE resources. They are exposed to the user only by direct accessing PE resources. These have a size of 8 bytes:

```
BYTE    fVirt;
BYTE    pad0;
WORD    key;
WORD    cmd;
WORD    pad1;
```

3. Accelerators in the Win32 API. These are exposed to the user by the `CopyAcceleratorTable` and `CreateAcceleratorTable` functions in the Win32 API. These have a size of 6 bytes:

```
BYTE    fVirt;
BYTE    pad0;
WORD    key;
WORD    cmd;
```

Why two types of accelerators in the Win32 API? We can only guess, but my best bet is that the Win32 resource compiler can/does not handle struct packing. Win32 ACCEL is defined using `#pragma(2)` for the compiler but without any packing for RC, so it will assume `#pragma(4)`.

Doing A Hardware Trace

The primary reason to do this is to reverse engineer a hardware device for which you don't have documentation, but can get to work under Wine.

This lot is aimed at parallel port devices, and in particular parallel port scanners which are now so cheap they are virtually being given away. The problem is that few manufactures will release any programming information which prevents drivers being written for Sane, and the traditional technique of using DOSemu to produce the traces does not work as the scanners invariably only have drivers for Windows.

Presuming that you have compiled and installed wine the first thing to do is to enable direct hardware access to your parallel port. To do this edit `config` (usually in `~/.wine/`) and in the ports section add the following two lines

```
read=0x378,0x379,0x37a,0x37c,0x77a
write=0x378,x379,0x37a,0x37c,0x77a
```

This adds the necessary access required for SPP/PS2/EPP/ECP parallel port on LPT1. You will need to adjust these number accordingly if your parallel port is on LPT2 or LPT0.

When starting wine use the following command line, where `XXXX` is the program you need to run in order to access your scanner, and `YYYY` is the file your trace will be stored in:

```
wine -debugmsg +io XXXX 2> >(sed 's/^[^:]*:io:[^ ]* //' > YYYY)
```

You will need large amounts of hard disk space (read hundreds of megabytes if you do a full page scan), and for reasonable performance a really fast processor and lots of RAM.

You will need to postprocess the output into a more manageable format, using the **shrink** program. First you need to compile the source (which is located at the end of this section):

```
cc shrink.c -o shrink
```

Use the **shrink** program to reduce the physical size of the raw log as follows:

```
cat log | shrink > log2
```

The trace has the basic form of

```
XXXX > YY @ ZZZZ:ZZZZ
```

where `XXXX` is the port in hexadecimal being accessed, `YY` is the data written (or read) from the port, and `ZZZZ:ZZZZ` is the address in memory of the instruction that accessed the port. The direction of the arrow indicates whether the data was written or read from the port.

```
> data was written to the port
< data was read from the port
```

My basic tip for interpreting these logs is to pay close attention to the addresses of the IO instructions. Their grouping and sometimes proximity should reveal the presence of subroutines in the driver. By studying the different versions you should be able to

work them out. For example consider the following section of trace from my UMAX Astra 600P

```
0x378 > 55 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > aa @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
0x378 > 00 @ 0297:01ec
0x37a > 05 @ 0297:01f5
0x379 < 8f @ 0297:01fa
0x37a > 04 @ 0297:0211
```

As you can see there is a repeating structure starting at address 0297:01ec that consists of four io accesses on the parallel port. Looking at it the first io access writes a changing byte to the data port the second always writes the byte 0x05 to the control port, then a value which always seems to 0x8f is read from the status port at which point a byte 0x04 is written to the control port. By studying this and other sections of the trace we can write a C routine that emulates this, shown below with some macros to make reading/writing on the parallel port easier to read.

```
#define r_dtr(x)      inb(x)
#define r_str(x)      inb(x+1)
#define r_ctr(x)      inb(x+2)
#define w_dtr(x,y)    outb(y, x)
#define w_str(x,y)    outb(y, x+1)
#define w_ctr(x,y)    outb(y, x+2)

/* Seems to be sending a command byte to the scanner */
int udpp_put(int udpp_base, unsigned char command)
{
    int loop, value;

    w_dtr(udpp_base, command);
    w_ctr(udpp_base, 0x05);

    for (loop=0; loop < 10; loop++)
        if ((value = r_str(udpp_base)) & 0x80)
        {
            w_ctr(udpp_base, 0x04);
            return value & 0xf8;
        }

    return (value & 0xf8) | 0x01;
}
```

For the UMAX Astra 600P only seven such routines exist (well 14 really, seven for SPP and seven for EPP). Whether you choose to disassemble the driver at this point

to verify the routines is your own choice. If you do, the address from the trace should help in locating them in the disassembly.

You will probably then find it useful to write a script/perl/C program to analyse the logfile and decode them further as this can reveal higher level grouping of the low level routines. For example from the logs from my UMAX Astra 600P when decoded further reveal (this is a small snippet)

```
start:
put: 55 8f
put: aa 8f
put: 00 8f
put: 00 8f
put: 00 8f
put: c2 8f
wait: ff
get: af,87
wait: ff
get: af,87
end: cc
start:
put: 55 8f
put: aa 8f
put: 00 8f
put: 03 8f
put: 05 8f
put: 84 8f
wait: ff
```

From this it is easy to see that `put` routine is often grouped together in five successive calls sending information to the scanner. Once these are understood it should be possible to process the logs further to show the higher level routines in an easy to see format. Once the highest level format that you can derive from this process is understood, you then need to produce a series of scans varying only one parameter between them, so you can discover how to set the various parameters for the scanner.

The following is the `shrink.c` program:

```
/* Copyright David Campbell <campbell@torque.net> */
#include <stdio.h>
#include <string.h>

int main (void)
{
    char buff[256], lastline[256] = "";
    int count = 0;

    while (!feof (stdin))
    {
        fgets (buff, sizeof (buff), stdin);
        if (strcmp (buff, lastline))
        {
            if (count > 1)
                printf ("# Last line repeated %i times #\n", count);
            printf ("%s", buff);
            strcpy (lastline, buff);
            count = 1;
        }
        else count++;
    }
    return 0;
}
```


Chapter 13. Porting Wine to new Platforms

This document provides a few tips on porting Wine to your favorite (UNIX-based) operating system.

Porting Wine to new Platforms

Why `#ifdef MyOS` is probably a mistake.

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

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

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

- existence of a header file
- existence of a library function
- existence of libraries
- bugs in header files, library functions, the compiler, ...

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

There *are* exceptions to the "avoid `#ifdef MyOS`" rule. Wine, for example, needs the internals of the signal stack -- that cannot easily be described in terms of features. Moreover, you can not use autoconf's `HAVE_*` symbols in Wine's headers, as these may be used by Winelib users who may not be using a `configure` script.

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

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

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

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

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

Now you can change

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

to

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

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

You will also need to add `#undef HAVE_FOO_H` (etc.) to `include/config.h.in`. Finish up with **make configure** and **./configure**.

MyOS doesn't have the `bar` function!

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

Secondly, you will also need to add `#undef HAVE_BAR` to `include/config.h.in`. The next step depends on the nature of the missing function.

Case 1:

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

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

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

Case 2:

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

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

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

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

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

Chapter 14. Consoles in Wine

As described in the Wine User Guide's CUI section, Wine manipulates three kinds of "consoles" in order to support properly the Win32 CUI API.

The following table describes the main implementation differences between the three approaches.

Table 14-1. Function consoles implementation comparison

Function	Bare streams	Wineconsole & user backend	Wineconsole & curses backend
Console as a Win32 Object (and associated handles)	No specific Win32 object is used in this case. The handles manipulated for the standard Win32 streams are in fact "bare handles" to their corresponding Unix streams. The mode manipulation functions (GetConsoleMode / SetConsoleMode) are not supported.	Implemented in server, and a specific Winelib program (wineconsole) is in charge of the rendering and user input. The mode manipulation functions behave as expected.	Implemented in server, and a specific Winelib program (wineconsole) is in charge of the rendering and user input. The mode manipulation functions behave as expected.
Inheritance (including handling in CreateProcess of CREATE_DETACHED, CREATE_NEW_CONSOLE flags).	Not supported. Every process child of a process will inherit the Unix streams, so will also inherit the Win32 standard streams.	Fully supported (each new console creation will be handled by the creation of a new USER32 window)	Fully supported, except for the creation of a new console, which will be rendered on the same Unix terminal as the previous one, leading to unpredictable results.
ReadFile / WriteFile operations	Fully supported	Fully supported	Fully supported
Screen-buffer manipulation (creation, deletion, resizing...)	Not supported	Fully supported	Partly supported (this won't work too well as we don't control (so far) the size of underlying Unix terminal)
APIs for reading/writing screen-buffer content, cursor position	Not supported	Fully supported	Fully supported
APIs for manipulating the rendering window size	Not supported	Fully supported	Partly supported (this won't work too well as we don't control (so far) the size of underlying Unix terminal)

Function	Bare streams	Wineconsole & user backend	Wineconsole & curses backend
Signaling (in particular, Ctrl-C handling)	Nothing is done, which means that Ctrl-C will generate (as usual) a <code>SIGINT</code> which will terminate the program.	Partly supported (Ctrl-C behaves as expected, however the other Win32 CUI signaling isn't properly implemented).	Partly supported (Ctrl-C behaves as expected, however the other Win32 CUI signaling isn't properly implemented).

The Win32 objects behind a console can be created in several occasions:

- When the program is started from wineconsole, a new console object is created and will be used (inherited) by the process launched from wineconsole.
- When a program, which isn't attached to a console, calls `AllocConsole`, Wine then launches wineconsole, and attaches the current program to this console. In this mode, the `USER32` mode is always selected as Wine cannot tell the current state of the Unix console.

Please also note, that starting a child process with the `CREATE_NEW_CONSOLE` flag, will end-up calling `AllocConsole` in the child process, hence creating a wineconsole with the `USER32` backend.

Chapter 15. How to do regression testing using CVS

A problem that can happen sometimes is 'it used to work before, now it doesn't anymore...'. Here is a step by step procedure to try to pinpoint when the problem occurred. This is *NOT* for casual users.

1. Get the "full CVS" archive from winehq. This archive is the CVS tree but with the tags controlling the versioning system. It's a big file (> 40 meg) with a name like full-cvs-<last update date> (it's more than 100mb when uncompressed, you can't very well do this with small, old computers or slow Internet connections).

2. untar it into a repository directory:

```
cd /home/gerard
tar -zxvf full-cvs-2003-08-18.tar.gz
mv wine repository
```

3. extract a new destination directory. This directory must not be in a subdirectory of the repository else **cv**s will think it's part of the repository and deny you an extraction in the repository:

```
cd /home/gerard
mv wine wine_current (-> this protects your current wine sandbox, if any)
export CVSROOT=/home/gerard/repository
cvs -d $CVSROOT checkout wine
```

Note that it's not possible to do a checkout at a given date; you always do the checkout for the last date where the full-cvs-xxx snapshot was generated.

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

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

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

4. you will have now in the ~/wine directory an image of the CVS tree, on the client side. Now update this image to the date you want:

```
cd /home/gerard/wine
cvs -d $CVSROOT update -D "2002-06-01 CST"
```

The date format is YYYY-MM-DD HH:MM:SS. Using the CST date format ensure that you will be able to extract patches in a way that will be compatible with the wine-cvs archive <http://www.winehq.org/hypermail/wine-cvs¹>

Many messages will inform you that more recent files have been deleted to set back the client cvs tree to the date you asked, for example:

```
cvs update: tsx11/ts_xf86dga2.c is no longer in the repository
```

cvs update is not limited to upgrade to a *newer* version as I have believed for far too long :-)

5. Now proceed as for a normal update:

```
./configure  
make depend && make
```

If any non-programmer reads this, the fastest method to get at the point where the problem occurred is to use a binary search, that is, if the problem occurred in 1999, start at mid-year, then if the problem is already here, back to 1st April, if not, to 1st October, and so on.

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

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

```
cvs -d $CVSROOT update -D "2002-06-01 15:17:25 CST"
```

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

6. If you find the patch that is the cause of the problem, you have almost won; report about it to Wine Bugzilla² or subscribe to wine-devel and post it there. There is a chance that the author will jump in to suggest a fix; or there is always the possibility to look hard at the patch until it is coerced to reveal where is the bug :-)

Notes

1. <http://www.winehq.org/hypermail/wine-cvs>
2. <http://bugs.winehq.org/>