Subject: WinAPI UNICODE question
Posted by cbpporter on Sat, 15 Nov 2008 10:34:55 GMT
View Forum Message <> Reply to Message

I'm trying to obtain a version of Core that does not expose one single bit of platform specific code at an interface level (but in implementation can use as much as needed), including WinAPI. This is not that easy. Simply removing include <windows.h> would lead to days of tracking down missing definitions.

So I started filtering windows.h, to make the task a lot easier. I also noticed some unneeded headers this way and a couple of cases where I think there ares mall errors related to "A" and "W" suffixes. Let me say that it is impressive how few WinAPI function U++ does actually use .

And this is where is stumbled across my question. For a given function f, under WinAPI there is a fA version, which works on ANSI strings, and a fW function which works on Unicode strings. If you want you application to be ansi, you leave the macro UNIDODE undefined, and f will automatically be translated to fA through macro substitution. By declaring UNICODE, it will get translated to fW. Unicode versions work only starting with windows 2000, but you can install some .dlls under Win95/98 to handle a subset of Unicode. Did I understand this correctlly?

So there are tree options:
1. Leave UNICODE undefined and get apps that run on all systems.
2. Define UNICODE and only run on Win2000 and 95/98 if dll is available.
3. Detect if Unicode support is available, and then call A or W version of functions manually depending on case.

I really can't figure out which one U++ uses. If I remove the #define SendMessage SendMessageA (for example), I get compilation error, which hints that UNICODE is undefined and we are using case number one. Also, there are some explicit cases where a W version is called, which makes me think that we are still using case 1 as a compilation environment, but we are trying to achieve functionality of 2/3 with the explicit calls of W variants in some cases.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sun, 16 Nov 2008 00:10:49 GMT
View Forum Message <> Reply to Message

After two unsuccessful attempts which left me with an uncompilable mess and I had to revert, I managed to get some good starting results.

As a first step I decided to move the whole block that prepares for the inclusion and includes <windows.h> as close to the end of Core.h as possible. And I got pretty far: only Path.h, Gtypes.h, Color.h, Lang.h, Xmlize.h and Win32Util.h must be included after windows.h. All other headers from Core have interfaces that are independent of windows.h, but I had to do some trivial changes in a lot of places. Most of the changes will even result in the same binary after compilation, so compatibility issues should be inexistent.

I compiled TheIDE with both MINGW and MSC in DEBUG and Optimal mode, ran it and

everything seems to be OK. Now I have to find a way to handle the last 6 headers...

Subject: Re: WinAPI UNICODE question
Posted by mirek on Sun, 16 Nov 2008 09:19:15 GMT
View Forum Message <> Reply to Message

I wonder a little bit, WHY?

To answer your question, we are using "3." - call 'W' version if running >=WinNT, 'A' version otherwise.

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sun, 16 Nov 2008 10:33:20 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 16 November 2008 11:19I wonder a little bit, WHY?

Well there are multiple possible answers.

First, it is problem of principle. U++ is my platform of choice for at least the next 2 years, and I want it to be THE platform, not just a platform combined with another platform (WinAPI) and another platform (various STD libs). So following this principle I don't want to wind up with a lot of useless stuff (which I will never ever use and am not able to use if I want cross-platform behavior) just for including a .h from U++.

Second, a practical consideration. Since U++ includes a lot of unnecessary and most importantlly platform dependent stuff, and doesn't provide all capabilities under all platforms, and since I use Windows as a primary development platform (and periodically compile under Linux), I end d up using unintentionally a lot of crap that simply does not compile under Linux. Now I have the nice situation of Windows version being OK and close to beta, and Linux one crippled for an unpredictably long time. It is OK to use it if I make a conscious choice (i.e. include a system dependent header), it's not OK if U++ allow me to compile that stuff by default.

Third, it is a principle of giving the precompiler less to work. Under MSC8, just for including Core.h I get about 240.000 lines of code that are precompiled. Time that by the number of packages for BLITZ builds and the number of compilation units for non BLITZ, and you do get a hefty sum. Sure, the by removing all the stuff and winding up with an about 25.000 line precompilation I won't be saving any lives, not even power bill (well maybe a completely unimportant sum), but still...

So in short:
#include <Core/Core.h>
CONSOLE_APP_MAIN
{

HWND hwnd;
}
Must not compile under Win32. The same goes if I replace HWND with pthread under Linux in a MT build. Preferably not even strcpy would compile without including <string.h>, but standard lib is stable enough to allow it (even though I can easily break cross-platform compatibility by calling some public functions that don't respect the standards, but I'm not going to clean up standard C library also).

Quote:
To answer your question, we are using "3." - call 'W' version if running >=WinNT, 'A' version otherwise.

Mirek
OK, I think I understand now. And where version without "A" or "W" are called, "A" versions are selected.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sun, 16 Nov 2008 11:08:38 GMT
View Forum Message <> Reply to Message

PS: Could I interest you in a couple of small changes that would leave Core compatible with both current code base and what I'm trying to achieve?

I'll use the first one as an example: in Defs.h, line 227,
typedef WCHAR wchar;
should be
typedef wchar_t wchar;
Actually this is what WCHAR does, so there should be absolutely no issues. Yet at this early stage, windows.h is not needed, so it would be IMO better to not rely on it.

Subject: Re: WinAPI UNICODE question
Posted by mirek on Sun, 16 Nov 2008 18:03:54 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sun, 16 November 2008 06:08PS: Could I interest you in a couple of small changes that would leave Core compatible with both current code base and what I'm trying to achieve?

I'll use the first one as an example: in Defs.h, line 227,
typedef WCHAR wchar;
should be
typedef wchar_t wchar;
Actually this is what WCHAR does, so there should be absolutely no issues. Yet at this early stage, windows.h is not needed, so it would be IMO better to not rely on it.

We need(ed) wchar to be compatible with win32 API. Using WCHAR in typedef is IMO safer way how to achieve it.

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sun, 16 Nov 2008 21:40:55 GMT
View Forum Message <> Reply to Message

Well, I know that, but WCHAR is not available at interface level if I want to achieve my goal. Anyway, in windows.h we have "typedef wchar_t WCHAR" and this is very unlikely to change. Such aggressive assumptions are needed, like replacing HANDLE with void*. Their number is quite limited though,and I think I had to replace about 5 different types with their underlying type.

Anyway, I'm down to my last two headers. Win32Util is easy, but Path is hard.

Other elements that are giving me a little trouble are the two .dli. Why was the .dli mechanism used here. The functions seem complicatedly normal Win32 functions which where available already. Is is for the structured call mechanism supplied by .dli or is there another reason?

Also, BLITZ is giving me funny issues. After an hour (or whatever the threshold is) has passed since last edit, I suddenlly get duplicate definition errors). It's actually easy to correct something like this,and even to not make them in the first place, but I'm not used to be on the lookout for them.

Subject: Re: WinAPI UNICODE question
Posted by mirek on Mon, 17 Nov 2008 16:14:19 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sun, 16 November 2008 16:40Well, I know that, but WCHAR is not available at interface level if I want to achieve my goal.

A couple of years ago, I was trying something similar. I got stuck with all those types like "HWND" etc... - it is mess to avoid using them in as class members... I guess, WCHAR is exactly the same story.

Moreover, you need to expose them in public interface too from time to time. We definitely need methods returning HWND, with LPARAM / WPARAM parameters etc...

Quote:
Such aggressive assumptions are needed, like replacing HANDLE with void*.

Exactly, that is where I have ended I have came to conculsion that it is not worth the trouble.

Quote:
Other elements that are giving me a little trouble are the two .dli. Why was the .dli mechanism used here. The functions seem complicatedly normal Win32 functions which where available already. Is is for the structured call mechanism supplied by .dli or is there another reason?

A/W versions.

Quote:
Also, BLITZ is giving me funny issues. After an hour (or whatever the threshold is) has passed since last edit, I suddenlly get duplicate definition errors).

You have not expected free lunch, right?

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Tue, 18 Nov 2008 11:41:57 GMT
View Forum Message <> Reply to Message

Another possible solution: introduce a flag and depending on it use either WinAPI types or their equivalent types.

```
#idndef flagNOPS
typedef WCHAR UWCHAR;
#else
typedef wchar_t UWCHAR;
#endif
```

The different name is needed to differentiate between the types if the flag is not set. All functions with platform dependent parameters or return values will use the Uxxxx variant, which will default to the WinAPI version by default, but if someone want the other version, they simply add the flag. I still have about 3 other solutions for that problem, but these two are the prettiest.

Except the issues of type names, there are two more:
1. Some macro's defined in windows.h are used which will no longer be available at that point. This can easily be solved with introducing either a copy of that macro with a different name (so that later you can include windows.h if needed or even better, by using an inline function, i.e. replacing HIWORD with HighWord inline function.

2. The use of the operators that convert from Point to POINT must be replaces with the uses of a function with exactly the same body as the operator, but which is an inline function that takes a Point parameter rather than an operator. I've done this in my code base already and only accrued in a few places, like in Draw.

And that's about it. With types not taken from windows.h, not using the macros from there and the issue with U++ geometric types, one can easily remove the windows reference and add it only to really low level compilation units.

I'm pretty much done with code and preparing to tackle Draw, I'll still hand around Core a little more to test and make sure that everything is OK. I also went over the packages from Bazaar (not the latest from SVN, the ones from 2008.1) and there are no issues here either. I was really expecting the docking functionality to at least not compile, but I'm glad to say I was wrong.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sat, 22 Nov 2008 10:36:02 GMT
View Forum Message <> Reply to Message

OK, so I was about 20% through Draw and I was starting to feel the good old "this is getting really old really fast". For every WinAPI reference I have to replace the type with the underlying one, both in header and C++, and often I have to insert harmless but ugly casts. Following this method I can't see myself finishing this, not to speak about merging with SVN anytime someone commits. Auto merge is pretty smart, but it has it's limitations.

So I need a better solution and I found one: a variant of the pimpl idiom. All Hxxxxx WinAPI types are pointers, and they receive their memory from some source (so it bypasses theproblem with pimp: the need to allocate extra memory), so a forward declaration in in function declaration is more than sufficient. So all I have to do is insert a forward struct and a typedef in Core.h somewhere early, and thanks to the very permissive forward declaration and typedef rules provided by the C++ standard (stuff which I usually would consider better not to compile), I can leave pretty much all U++ header and .cpp files unmodified. This will make merges a lot easier.

So ignore all my previous suggestion (except the MACROs from WinAPI). I think this is the right way to handle this. Very few changes to existing code base, yet still platform independent interfaces.

It has the slight disadvantage that something like HINSTANCE h; will compile anywhere, but it's just a pointer and you can't really introduce platform dependency with an opaque pointer, unless you really want to.

It even maintains the 15% compilation time decrease under MSC when compiling Core tutorials. I need to get more packages done before I can say how this percent will change when compiling larger code base.

Now I have to undo my changes and experiment with this approach. If somebody is interested, I can post here a Win32 independent version of Core once I consider it stable, and later Draw.

Subject: Re: WinAPI UNICODE question
Posted by mirek on Sun, 23 Nov 2008 14:49:28 GMT
View Forum Message <> Reply to Message

I keed my eye on this effort, but I am neither optimisitic or enthusiastic.

Been there, tried that. In the end, the main problem is that you are still on host platform a need to do platform specific stuff here and there. Means, among other things, you cannot afford name clashes with both X11 and Win32 API.

Given this, including platform API headers is maybe a little bit unelegant, but the most straightforward solution. The only real disadvantage I can see is longer build time.

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Sun, 23 Nov 2008 16:39:13 GMT
View Forum Message <> Reply to Message

Well there is one small problem: some function names get overridden with a different name. For example, Upp::GetModuleFileName becomes Upp::GetModuleFileNameA. This is because of the macros from windows.h, which happily traverse namespace borders. I don't know if this can pose a problem, but it could give rise to surprises when linking.

As for progress, I covered CtrlLib and it's dependencies. Next: TheIDE .

Testing with Bombs example package, I can say that the 15% compilation time decrease remains valid for MSC (I test by giving a rebuild all command), both for total time,and per package basis. Nothing to get too excited over, but since it's a "free" gain, I don't see why I shouldn't be happy about it.

Subject: Re: WinAPI UNICODE question
Posted by bytefield on Sun, 23 Nov 2008 17:46:32 GMT
View Forum Message <> Reply to Message

Is your work related to "cleaning up" the Upp from unwanted dependencies, functions conflict, "old" not so good design, etc. or it is a derivation of Upp? If not, when your modifications will be available in Upp sources?

Subject: Re: WinAPI UNICODE question
Posted by mirek on Sun, 23 Nov 2008 20:32:17 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sun, 23 November 2008 11:39Well there is one small problem: some function names get overridden with a different name. For example, Upp::GetModuleFileName becomes Upp::GetModuleFileNameA. This is because of the macros from windows.h, which happily traverse namespace borders. I don't know if this can pose a problem, but it could give rise to surprises when linking.

Yes, I am aware about that. IMO, it is not really a problem.

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Mon, 24 Nov 2008 00:04:57 GMT
View Forum Message <> Reply to Message

bytefield wrote on Sun, 23 November 2008 19:46Is your work related to "cleaning up" the Upp from unwanted dependencies, functions conflict, "old" not so good design, etc. or it is a derivation of Upp? If not, when your modifications will be available in Upp sources?
It is related to exposing as little as possible as possible from the target platform API at an interface level. It only allows platform specific code (right now just WinAPI, but later can be done for Linux also, I think even easier) to not compile or be useful for client of U++ API. Also, I removed some unwanted external headers. All WinAPI functions are hidden, and only some types are available, but these types are available independent of platform. You can use a HWND under Linux, and both under Linux and Windows it will still be completely useless for you. It will just be a pointer type, without any way to use it in a platform dependent way (unless you go out of your way to cast it to something evil, but like C++ in general, my approach protects you from accident, not from intention). So this raises the question: how to deal with the parts of U++ that need Windows API. Well I made Core.h platform independent and added a CorePS.h that must be included in platform dependent compilation units. The number of compilation units that need platform specific code is very small, thus the 15% compilation time decrease. Except the new header that is included, and some small API differences that are applied here and there, the rest of the U++ code base remains unmodified. I'm not expecting this to be used in the official release, but to get it to a shape where 99% of it is identical to official U++, the rest of 1% is handled by auto merge, and the generated binary from compilation is the same.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Mon, 24 Nov 2008 00:06:27 GMT
View Forum Message <> Reply to Message

luzr wrote on Sun, 23 November 2008 22:32cbpporter wrote on Sun, 23 November 2008 11:39Well there is one small problem: some function names get overridden with a different name. For example, Upp::GetModuleFileName becomes Upp::GetModuleFileNameA. This is because of the macros from windows.h, which happily traverse namespace borders. I don't know if this can pose a problem, but it could give rise to surprises when linking.

Yes, I am aware about that. IMO, it is not really a problem.

Mirek

Can be solved with a simple undef and calling "A" version explicitly. This is what I used.

Subject: Re: WinAPI UNICODE question
Posted by mirek on Wed, 26 Nov 2008 18:41:44 GMT
View Forum Message <> Reply to Message

Quote:
Can be solved with a simple undef and calling "A" version explicitly. This is what I used.

Actually, a good idea - we might need that when A++ parses macros

Mirek

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Fri, 05 Dec 2008 04:02:37 GMT
View Forum Message <> Reply to Message

cbpporter wrote on Sun, 23 November 2008 18:39 Next: TheIDE .

Done! I'm running it right now. I'll have to test it carefully to see if everything is OK. I had to make a compromise for ide and ide\Debuggers, but these two packages are very high in the hierarchy and pretty platform dependent, so I guess it's not that much of a compromise.

Now comes the fun part and I'll have to merge my local modifications with SVN updates so I can keep up to date. I'm hopping for few conflicts .

Here is the list of macros that I found troublesome from windows.h:

```
#undef min
#undef max
#undef GetModuleFileName
#undef GetWindowsDirectory
#undef GetFileTitle
#undef DrawText
#undef LoadString
#undef DeleteFile
#undef GetTempFileName
#undef LoadLibrary
```

#undef GetCurrentDirectory

DeleteFile and GetTempFile are interesting. By using the "A" variant, I'm not sure if you can delete a file with Unicode chars. I'll have to check it out.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Tue, 09 Dec 2008 08:41:46 GMT
View Forum Message <> Reply to Message

I am going to give now a concrete example with attached modified files. I've chosen Color.h and Color.cpp to illustrate some points because they are so easy and universally used throughout my modified files.

What I done here is added inline copies of the function GetRValue to RGB with different names and used them. GetRValue and friends are declared as macros in windows.h and and as inline functions under POSIX so it would seem that we are covered in regards to cross-platform compatibility. Unfortunately, this is not the case.

The biggest problem is that you can not write code without having using namespace Upp somewhere before. If I try to use Upp::GetRValue, it will compile and work fine under Linux, but not under Windows. I havent't encountered this problem with exactlly this function, but I did with other functions which I'm going to touch in later posts (I'm looking at GetTickCount especially).

Another problem is that under Linux, you can get the adress of such functions,while under Windows the same code will not compile again.

This is why I'va chosen to introduce new names. If you don't like the solution, GetRValue and friends could be undefed and defined as macros under all platforms and in Upp namespace, not just under POSIX. But this could lead to name clashes if someone later includes windows.h again.

File Attachments
1) Core.zip, downloaded 388 times

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Wed, 10 Dec 2008 19:41:43 GMT
View Forum Message <> Reply to Message

I'm uploading here a first version. It is more of a proof of concept than anything else at the moment. It is not as clean as I wanted it to be, because midway I decided that compatibility is more important right now than completely clean code. Also, because of the 2MB maximum attachment size, I could barely include enough to compile something like UWord with striped out documentation. I need to upload to rapid share or something. I haven't tested all the packages, but almost all will compile and work fine under Windows at least. Some of the ones that don't compile don't compile under unmodified SVN also, so I guess it's not me.

Subject: Re: WinAPI UNICODE question
Posted by cbpporter on Thu, 11 Dec 2008 09:04:42 GMT
View Forum Message <> Reply to Message

I've uploaded the whole thing on RapidShare. Only 10 downloads are allowed. I'm going to remove the attachment from previous message.

The packages that don't compile: Docedit, GLCtrl, ide/VectorDes, Ole/Ctrl, rw, TDraw and VectorDes. Number of modified files: 150.