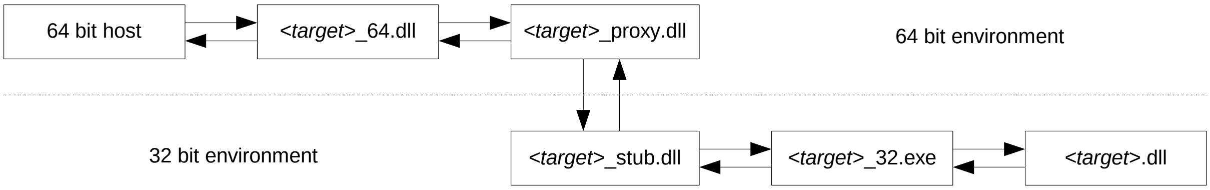
DLLWrapper Manual

Introduction

Thank you for using DLLWrapper, a tool that gives you the ability to use 32 bit DLLs in a 64 bit host environment and vice versa. It allows you to keep on using software and hardware you bought for a huge pile of money in the past while switching to a 64 bit Windows operating system for instance even when there are no 64 bit DLLs available to access your old software or hardware.

The only input that the tool needs is the original DLL that it should wrap and a C header file describing the interface functions of the DLL. Ideally a suitable header file will have been delivered by the vendor of the DLL – if this is not the case, you'll have to write a header file by yourself by extracting the necessary information from some form of documentation that should have been provided by the vendor of the DLL at least.

In order to wrap for instance a 32 bit target DLL so that it can be used again in a 64 bit environment by a host program like the 64 bit version of Matlab for example, the tool will generate a 64 bit wrapper DLL providing the same functions as the original 32 bit target DLL as well as a 32 bit wrapper EXE which will link the original DLL and two proxy / stub DLLs for inter process communication marshalling. When the 64 bit host program loads the wrapper DLL and starts calling a function provided by the DLL, the function call will be mapped to the invocation of a COM object method passing the barrier between the 64 bit process environment of the host application and the 32 bit process environment of the target DLL by executing the call in the environment of the wrapper EXE as shown below:



For the rest of this document it is assumed that you're able to read (and possibly write) C header files - if you're not at home in this, you should find someone who is... We'd like to emphasize that this tool is designed for **experts** with some **programming experience** as deepened understanding of the Windows operating system as well as procedure call and parameter passing mechanism are vital to achieve a smoothly running wrapper DLL.

Attention: as it is not possible to guarantee for any of a literally infinite number of combinations that DLLWrapper will generate error-free resulting software, you as the user are responsible for testing the correct behavior of the generated wrapper DLL in all of your use cases - we can't take that off your shoulders!

Installation

Before installing and using DLLWrapper you first have to install some necessary third-party build tools that DLLWrapper is relying on. All of those tools are available free-of-charge but due to legal reasons we weren't able to bundle them with DLLWrapper so you have to install those tools manually before. System requirement for DLLWrapper is a PC with a 64 bit Windows operating system like Windows Vista, Windows 7 or Windows 8 - it can't be used on a 32 bit Windows operating system. As it is deprecated we don't support Windows XP 64 bit, too.

At first you should install a version of the free VisualStudio Express Edition from Microsoft's web site http://www.visualstudio.com/downloads/download-visual-studio-vs - you can use either the 2010 version (which is the last version that can be installed under Windows Vista) or the 2012 or the 2013 version of VisualStudio Express. If you already have other editions of VisualStudio (Professional, Premium, Ultimate) installed, they should work, too, but we can't guarantee that.

If you've installed the 2010 version of VisualStudio Express, you'll have to install the Windows SDK 7.1 from Microsoft's web site http://www.microsoft.com/en-us/download/details.aspx?id=8279, too, (or if you prefer an ISO-Image from http://www.microsoft.com/en-us/download/details.aspx?id=8442, in which case you'll need the image GRMSDKX_EN_DVD.iso for the 64-Bit-SDK) as it includes the needed 64 bit C/C++ compiler and some other tools like the MIDL compiler which are not part of the VisualStudio 2010 Express Edition (but of the 2012 and 2013 version).

Please note that if you are installing the additional service pack for VisualStudio 2010 from http://www.microsoft.com/de-de/download/details.aspx?id=23691 (which we recommend especially if you experience some problems) *after* installing the SDK, this will accidentally remove the 64 bit compiler again. You'll then have to reinstall it using the hotfix that Microsoft provided under http://support.microsoft.com/kb/2519277/en-us for this installation bug.

If you're planning to distribute wrapping DLLs to some other people like colleagues or customers, you also need to install the installer generator InnoSetup from http://www.jrsoftware.org/isinfo.php.

After installing DLLWrapper with the respective option checked you'll find some real world examples in the public document folder in the directory DLL Wrapper Examples (simply follow the link created in the start menu). We have installed those files in this directory so that you or other users are able to build the examples even without admin rights. At the moment the following examples are provided:

Example	**Description**	**Web site**
K8055D	Wrapper to access Velleman's K8055D / VM110 USB Multi-I/O-Board from 64 bit	http://www.velleman.eu/products/view/?id=404998
PCSGU250	Wrapper to access Velleman's PCSGU250 oscilloscope / generator from 64 bit	http://www.velleman.eu/products/view/?id=377622
USBAXLA	Wrapper to access USBees AX oscilloscope / logic analyzer from 64 bit	http://www.usbee.com/ax.html
WrapperTestDll	Comprehensive test DLL for DLLWrapper simply writing function parameters into file	DLL (32 and 64 bit) and source files included

Due to legal reasons we couldn't bundle some of the example DLLs with DLLWrapper - if you like to build them, you'll have to install the respective third-party software from the given web site first.

Preparing a C header file

As host application with wrapper DLL on the one hand and wrapper EXE with target DLL one the other hand are running in separate processes with own address spaces, all data passed to and from the DLL functions has to be copied from one address space to the other. Fortunately, the Microsoft COM environment provides means to automatically generate code that will do the necessary coping including packing the data on one side and unpacking it on the other side for us – the so-called marshalling code. However, the tools generating this marshalling code need precise information about the size of the memory chunks to be copied.

If the DLL you'd like to wrap only provides functions with basic parameter and return types like int, float etc., all the necessary information for the marshalling is already present in a C header file describing the DLL interface and no modifications to the header file will be necessary in order to use it with DLLWrapper. If the DLL functions use pointers to pass arrays to and from the host application by reference, some modifications might be necessary however in order to provide information about the actual array size. Furthermore modifications might be necessary if a header file includes some nested headers so that they can be found by DLLWrapper. If you think that you might need to make modifications to a header file provided by your DLL vendor, we recommend that you'll make a local copy of this header file first before starting to modify it. If your DLL vendor didn't provide a C header file for the DLL at all, you'll have to write one by yourself in order to provide the necessary information to DLLWrapper by extracting the information from the DLL interface documentation.

DLLWrapper uses the following default assumptions regarding parameter passing when parsing a given header file:

1. All non-pointer and non-array parameters (that includes `struct` and `union` types) are pure inputs (this is not an assumption but rather a fact as C passes all first-level parameters by value) and therefore must be copied only from host application to target DLL.
2. All pointer parameters effectively (maybe hidden by some typedefs) pointing to basic data type `char` are assumed to be C strings with maximal length of MAX_STR+1 (this configuration parameter can be changed and passed via GUI or command line interface). Furthermore it is considered that the pointer could also be a NULL pointer.
3. All pointer parameters effectively (maybe hidden by some typedefs) pointing to basic data type `unsigned char` are assumed to be raw data buffers with maximal length of MAX_BUF) (this configuration parameter can be changed and passed via GUI or command line interface, too). Furthermore it is considered that the pointer could also be a `NULL` pointer.
4. All other pointers are assumed to point to exactly one object of the referred data type - that means it is NOT assumed, that pointer types other than in the two cases above are pointer to arrays by default. Furthermore it is considered that the pointer could also be a `NULL` pointer.
5. Arrays with a given size passed as function parameters (which are always passed by reference in C) are handled as such.
6. If a pointer points to a data type marked as `const`, it is assumed that this memory will be read-only for the function it is passed to and therefore it will only be copied from

host application to target DLL. If a pointer points to a data type NOT marked as const however, it is assumed that the called function will change the contents of the memory passed and therefore it will also be copied from target DLL to host application again after the call.

7. Pointers to memory chunks returned by a DLL function are assumed to be pointing to statically allocated memory inside the original function.

Especially if the functions described in your header file are using pointers to pass arrays from and to the host application with types different from char and unsigned char (which is quite common praxis in C), you'll have to modify the function prototype in order to provide the correct information for DLLWrapper (you have to get DLLWrapper to use assumption 5 instead of 4). In case you know from the DLL documentation that function foo is called with an integer array with a fixed number of 16 elements, you'll have to change

```
void foo (int *tab);
```

into

```
void foo (int tab[16]);
```

In case an array has a variable size depending on some other function parameters, you can change

```
void foo (int len, int *tab);
```

into

```
void foo (int len, int tab[len]);
```

In the rare case that the size of an array is neither fixed nor can be determent by other function parameters, a wrapping of this function is not possible with DLLWrapper. If you don't need the function anyway, you can simply delete it from the header file - otherwise you'll have a problem now...

For the sake of efficiency it is recommended to add the `const` modifier to pointer function parameters that are knowingly not modified by the called function. You should check your DLL documentation to extract this information as this will reduce the unavoidable overhand imposed by inter process communication significantly. If you retrieve from the documentation of your DLL that the function `foo` does not change the contents of the array `tab` passed to it for instance, you should change

```
void foo (int tab[16]);
```

into

```
void foo (const int tab[16]);
```

Generating a wrapper DLL

After you have prepared the necessary C header file, you can either use the command line version of DLLWrapper or a little GUI to generate a wrapper DLL for your given original target DLL. Both the command line interface and the GUI are described in detail below. In both cases you can pass an output directory in which the final products should be placed. For a DLL named <target>.dll with a corresponding header file <target>.h, the following products will be generated when wrapping a 32 bit DLL:

<target>_64.dll	64 bit wrapper DLL, replacement DLL for 32 bit target DLL
<target>_32.exe	32 bit wrapper EXE, replacement process to host 32 bit target DLL
<target>_stub.dll	32 bit marshalling code
<target>_proxy.dll	64 bit marshalling code
<target>_reg.bat	Batch file to register COM objects needs for inter process communication
<target>_unreg.bat	Batch file to unregister COM objects

When wrapping a 64 bit target DLL to be called from a 32 bit host application, the replacement DLL will be called <target>_32.dll while the replacement process will be called <target>_64.exe.

Please note: if you're using the trial version of DLLWrapper, the successful building of a wrapper will only be possible for thirty times - afterwards you'll have to buy a full license.

Using a wrapper DLL

After generating a wrapper DLL and the accompanying files the new COM objects realizing the inter process communication between 64 bit host application and 32 bit target DLL have to be registered with the Windows COM system. For that you'll have to execute the generated batch file <target>_reg.bat (or you can choose to directly register the COM objects after building a wrapper by using the respective GUI or command line option). Please note that you'll need at least temporary admin rights on your machine for that.

Afterwards the wrapper DLL can be used by a host application by simply copying it to a place where it can be found by the host application - you can either copy the DLL to the system32 subdirectory of your windows installation (in case it's a 64 bit DLL, otherwise you'll have to copy it to the syswow64 directory in case it's a 32 bit DLL) or into the directory where the exe file of the host application is located. Please note that you might need temporary admin rights on your machine for that. If your host application uses run time linkage instead of load time linkage it might be possible to provide a path directly pointing to the output directory used by DLLWrapper so that copying the wrapper DLL to some other directory might not be necessary at all.

Please note: if you're using the trial version of DLLWrapper, a generated wrapper DLL will **only run for 24 hours** after it has been build - afterwards it either has to be rebuild or you have to buy a full license ;)...

Furthermore it is possible to automatically generate an installation package for the built wrapper DLL using the corresponding option of the GUI or comand line interface in order to distributed it to other systems. This feature is not available with the trial version of DLLWrapper.

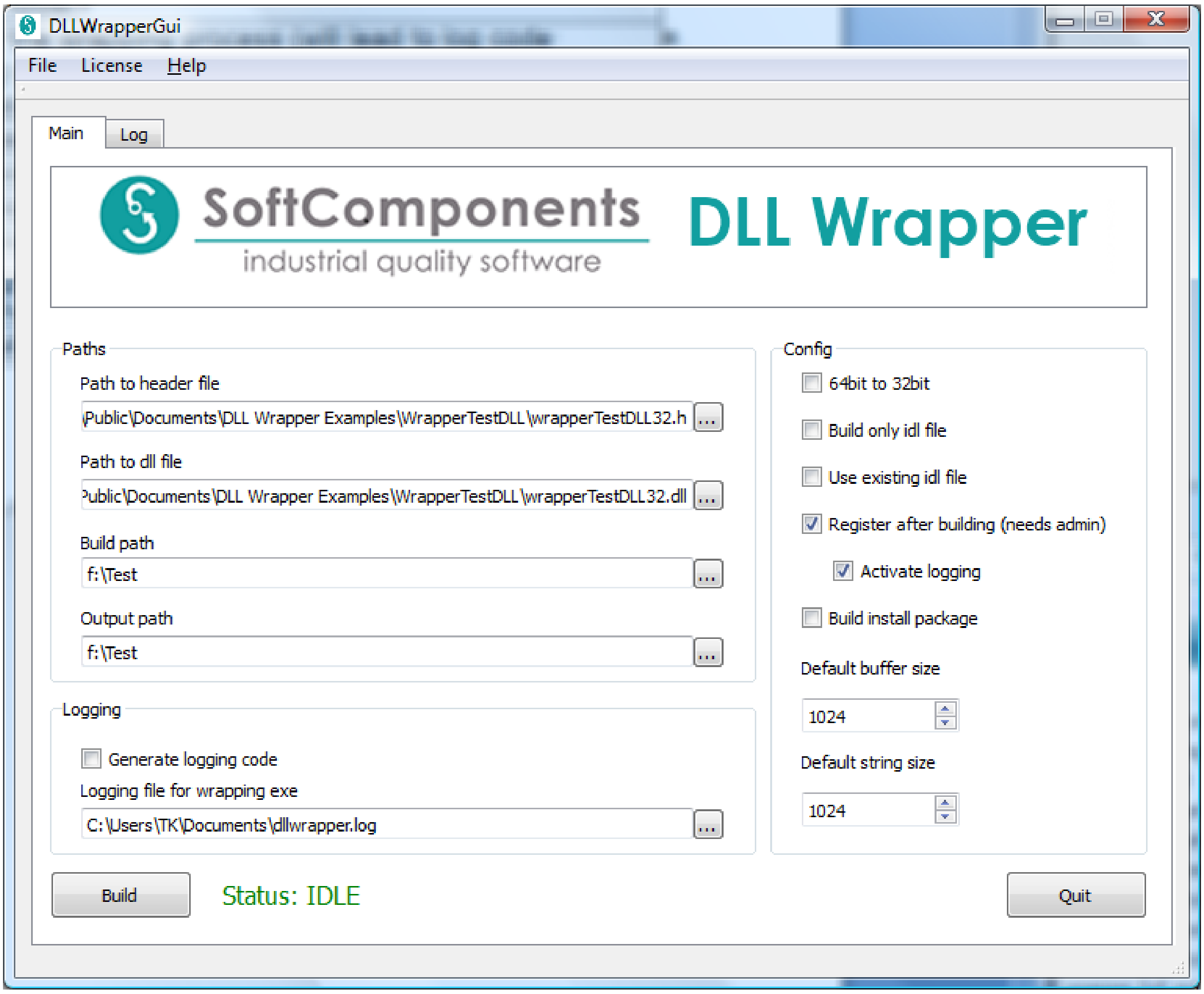
Command line interface

The command line version of DLLWrapper expects the name and the path of a header file describing the DLL interface to be wrapped as a command line parameter. Furthermore a number of additional command line options (which can be prefaced by either - or /) are supported:

H,?	a **h**elp page
V	**v**erbose mode, echo all invocations of external tools (useful to investigate problems with the build process due to failing tool invocations).
O=<path>	**o**utput path for products
B=<path>	**b**uild path for intermediates (intermediates visible only with full license)
D=<path>	explicit **D**LL file path and name, necessary if header file name is different from dll file name or placed in different folder
L=<path>	**l**og file path and name for the wrapping process (will lead to log code generation in the wrapping process which is very useful to investigate problems using the generated wrapper DLL).
I	generate **i**dl file only for manual modifications (possible only with full license)
E	generate products from **e**xisting idl file (possible only with full license)
X	e**x**change wrapping direction from 32 -> 64 to 64 -> 32 bit
R	**r**egister products after building
A	**a**ctivate logging directly (only together with option R)
P	build an installation **p**ackage (possible only with full license)
C=<size>	set the maximal size for **c**haracter strings (default 1024)
U=<size>	set the maximal size for data b**u**ffers (default 1024)

Interactive GUI

The interactive GUI provides the same features as the command line interface version of DLLWrapper but might offer a more intuitive way of using the tool. Below the "Main" tab of the GUI is depicted:



Paths

Path to header file: Public\Documents\DLL Wrapper Examples\WrapperTestDLL\wrapperTestDLL32.h

Path to dll file: Public\Documents\DLL Wrapper Examples\WrapperTestDLL\wrapperTestDLL32.dll

Build path: f:\Test

Output path: f:\Test

Logging

□ Generate logging code

Logging file for wrapping exe: C:\Users\TK\Documents\dllwrapper.log

Config

□ 64bit to 32bit

□ Build only idl file

□ Use existing idl file

☑ Register after building (needs admin)

☑ Activate logging

□ Build install package

Default buffer size: 1024

Default string size: 1024

Status: IDLE

In the "Paths" section of the GUI you can select or enter the path to and the name of the header file describing the DLL exports, the path to and the name of the target DLL that should be wrapped, the build path, where the intermediate files should be placed, and the output path, where the final products should be placed. Note that in the trail version of the tool, you can't select a build path because all intermediate files will be deleted right away during the build.

In the "Logging" section of the GUI you can activate the generation of logging code into the wrapper EXE server which is very helpful in finding problems with wrapped DLL functions (see the chapter "Troubleshooting" below). Furthermore you can select a path and a name for the logging file.

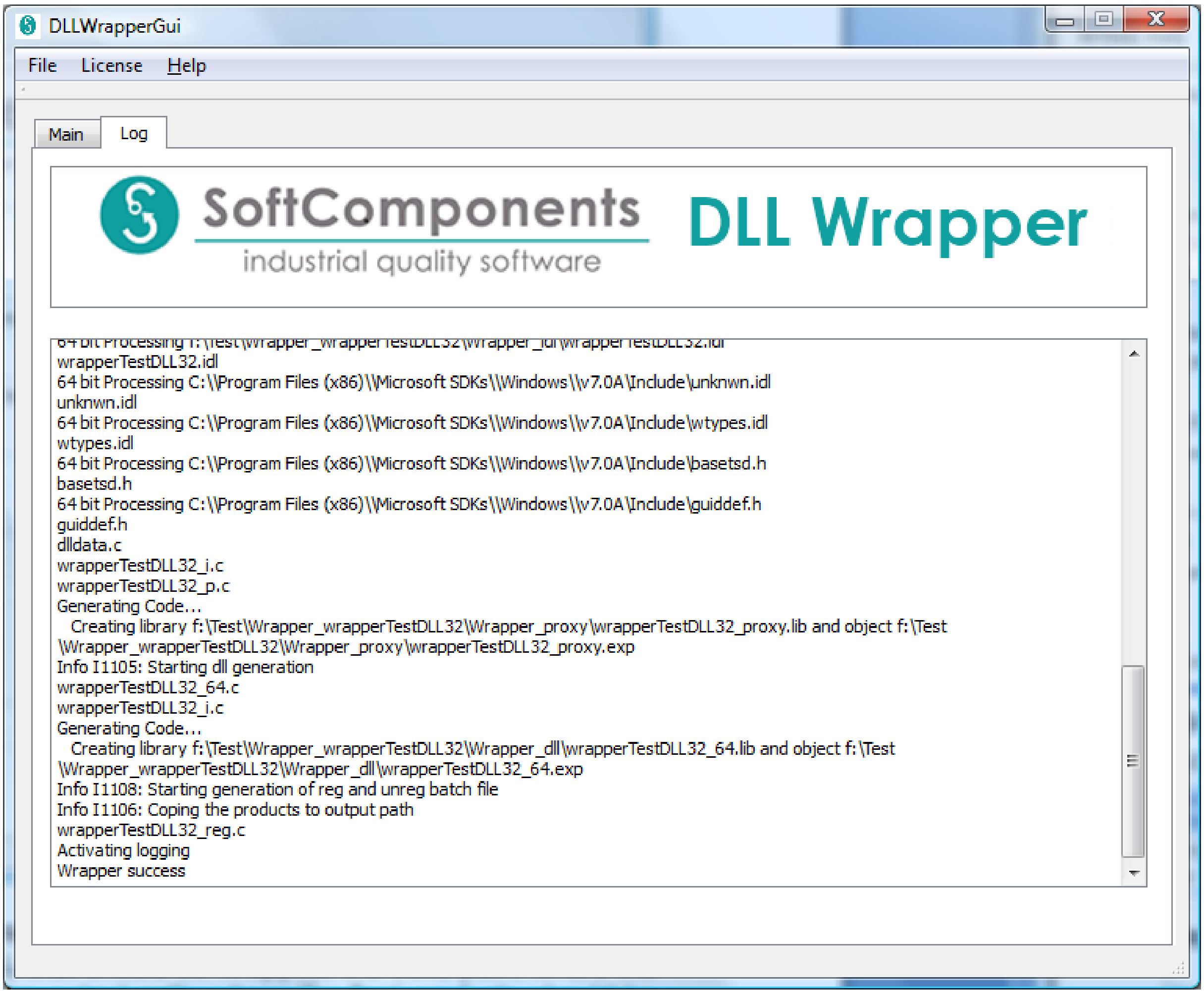
In the "Config" section you can change some settings for the wrapping procedure. You can reverse the wrapping direction to wrap 64 bit DLLs as 32 bit DLLs and you can enable the registration of the generated wrapper during the build in order to be able to test it directly afterwards. For the registration you'll need admin rights, however. Furthermore, if registration is enabled you can activate the logging right away, too.

With a fully licensed DLLWrapper you'll be able to stop the build process after the generation of the IDL file in order to be able to make manual modifications to it before continuing the build with an already existing IDL file. Please note that in this case the IDL file is expected to be located in the sub directory <target>_wrapper\<target>_idl in the build directory, into which the generated idl file will be placed initially - no different location can be selected.

Furthermore you'll be able to generate an installation package for the generated wrapper DLL in order to be able to distribute it to other systems. For this additional feature you'll have to install Inno Setup as external tool to generate installation packages, however.

Finally you can change the default maximal size for strings (MAX_STR) and buffers (MAX_BUF) in the "Config" section (see below for an explanation of these values).

In the "Log" tab depicted below you'll be able to see messages produced by DLLWrapper and the external tools invoked by DLLWrapper like the MIDL compiler and the C++ compiler:



```
64 bit Processing f:\Test\Wrapper_wrapperTestDLL32\Wrapper_idl\wrapperTestDLL32.idl
wrapperTestDLL32.idl
64 bit Processing C:\\Program Files (x86)\\Microsoft SDKs\\Windows\\v7.0A\Include\unknwn.idl
unknwn.idl
64 bit Processing C:\\Program Files (x86)\\Microsoft SDKs\\Windows\\v7.0A\Include\wtypes.idl
wtypes.idl
64 bit Processing C:\\Program Files (x86)\\Microsoft SDKs\\Windows\\v7.0A\Include\basetsd.h
basetsd.h
64 bit Processing C:\\Program Files (x86)\\Microsoft SDKs\\Windows\\v7.0A\Include\guiddef.h
guiddef.h
dlldata.c
wrapperTestDLL32_i.c
wrapperTestDLL32_p.c
Generating Code...
   Creating library f:\Test\Wrapper_wrapperTestDLL32\Wrapper_proxy\wrapperTestDLL32_proxy.lib and object f:\Test
\Wrapper_wrapperTestDLL32\Wrapper_proxy\wrapperTestDLL32_proxy.exp
Info I1105: Starting dll generation
wrapperTestDLL32_64.c
wrapperTestDLL32_i.c
Generating Code...
   Creating library f:\Test\Wrapper_wrapperTestDLL32\Wrapper_dll\wrapperTestDLL32_64.lib and object f:\Test
\Wrapper_wrapperTestDLL32\Wrapper_dll\wrapperTestDLL32_64.exp
Info I1108: Starting generation of reg and unreg batch file
Info I1106: Coping the products to output path
wrapperTestDLL32_reg.c
Activating logging
Wrapper success
```

Under the "File" menu you'll find two items to clear all settings or the reload the last settings that were stored when closing the tool the last time as well as one item to close the tool.

Under the "License" menu you'll find two items to buy or update to a full license for DLLWrapper. The "Buy license" item will lead you to our web site where you can directly purchase a full license for DLLWrapper. We'll deliver a license file to you via e-mail after receiving your payment and you'll be able to upgrade your installed trail version of DLLWrapper to a full featured version without any limitations by using the "Update license" item to select the license file you received and saved somewhere in your file system.

Finally under the "Help" menu you'll find one item to open this DLLWrapper user manual in the *.chm help file format used by recent Windows versions (a link to a PDF version of this document can be found in the start menu). Furthermore you'll find a menu item giving you information about you'll license type (trail or full) and our contact e-mail address.

Limitations

Handle-like function parameters

As DLLWrapper is trying to convert in-process function calls to a target DLL in the same process environment into out-of-process COM calls with two processes in different environments, there are some limitations regarding the indirect passing of memory. As there is no single address space any more between the wrapped target DLL and the host application, memory addresses passed directly and unchanged from the host application to the target DLL and vice versa wouldn't have a useful meaning in the respective other address space. However there are cases in which it is necessary to pass an unchanged memory address from the target DLL to the host application if this memory address is used as a kind of handle for later calls to other DLL functions for instance (see our USBAXLA example for the USBee AX Oscilloscope / Logic Analyzer, where such a case occurred). In this case you'll have to change the data type of the respective pointer parameters into `unsigned long` (if wrapping a 32 bit target dll) or `unsigned long long` (if wrapping a 64 bit target DLL). In the USBAXLA example we had to change the following function prototypes for instance

```
    extern unsigned long *MakeBuffer(unsigned long size);
    extern int DeleteBuffer(unsigned long *buffer);
```

into this

```
    extern unsigned long MakeBuffer(unsigned long size);
    extern int DeleteBuffer(unsigned long buffer);
```

as the pointer returned by the function `MakeBuffer` is used as a pointer to data but also as a handle for an allocated buffer which has to be passed to the function `DeleteBuffer` later on to free the buffer after usage.

In order to be able to access contents at the memory address passed by the target DLL in case that the address is not used as a handle exclusively but also use as a memory buffer for instance (as in the USBAXLA example), two additional generic function will always be generated into the wrapper DLL which can be used to directly read and write memory in the address space of the target DLL:

```
    void <target>_ReadMem (void *buffer, unsigned long addr,
                           unsigned long size);
    void <target>_WriteMem (unsigned long addr, void *buffer,
                            unsigned long size);
```

At this point the compatibility of the wrapper DLL with the target DLL is of course not 100 % and programs or scripts using the DLL functions that have been written for a 32 bit host environment will have to be adopted in this case. **Attention:** please take special care when using the above function to read and / or write memory in the target DLLs address space that you are providing a valid address and a valid length - if the memory access will fail due to invalid address and / or length, the wrapper EXE server will crash and the `ReadMem` / `WriteMem` function will return without result after an inter process communication timeout.

Memory returned by function

In some cases DLL functions are returning pointer to memory chunk allocated by the called function itself either statically (as global but function-local memory) or dynamically (from some form of memory heap). Such a pointer could be returned as a function result or by changing a pointer which address has been passed to the function as an argument (second level pointer or pointer to pointer). DLLWrapper assumes by default that all memory pointers returned by a function are pointing to statically allocated memory and it will allocated the same amount of memory statically in the replacement DLL as there is no way for DLLWrapper anyway to find out when and how dynamically allocated memory should be freed again.

In case of a function returning statically allocated memory it is again essential to provide precise information for DLLWrapper about the size of the memory returned. **Please note:** if this information is wrong, DLLWrapper might try to copy data beyond a valid memory chunk with might lead to an access violation in the wrapper EXE leading to a crash of it and a timeout of the inter process communication with the wrapper DLL. Returning a C character string like in the case

```
char *foo(void);
```

is rather uncritical as the wrapping code can determine the length of the string by searching the terminating null character - it only has to be insured that the configuration parameter MAX_STR is large enough for all possible strings passed as otherwise some strings might be truncated (but no access violation will occur). Return pointer to a data buffer like in the case

```
unsigned char *bar(void);
```

requires special care as the configuration parameter MAX_BUF has to be set exactly to the size of the returned buffer - if it's set to smaller value, the returned buffer will be truncated, if it's set to a larger value, an access violation might be the result. In general it's better to rewrite the function prototype to

```
unsigned char (*bar(void))[512];
```

in case the buffer is 512 bytes in size for instance. This is mandatory anyway for functions returning arrays of other types than `char` and `unsigned char` as DLLWrapper will assume that a pointer returned by the function will only point to a single type instance by default.

If a DLL function is actually returning dynamically allocated memory, the approach described in the section above has to be used and the pointer values should be passed unchanged directly from one a address space to the other as they are used as handles, too. In case such memory chunks were allocated from the default Windows process head (or a DLL function expects a memory chunk that has been allocated on the default process heap and will free it during the function call) two additional generic function will always be generated into the wrapping DLL which can be used to directly allocate and free memory on the default process heap in the address space of the target DLL:

```
unsigned long <target>_AllocMem (unsigned long size);
void <target>_FreeMem (unsigned long addr);
```

Unions

As there is usually no generic way to find out which element of a union is the "valid" one at a given point of time, unions can only be passed as chunks of memory that will be copied from one address space to the other without further marshalling. This implies that using unions with pointers in it will not work as the memory the pointer is pointing to is itself not copied and the value of the pointer will be of no meaning in the other address space. This is only a minor limitation as union are rarely used as DLL function parameters.

Bitfields

Bitfields are passed from one address space to the other simply by copying the contents of the memory cells they occupy. There is no guarantee that the bit order is compatible in this case (normally it should be), but using bitfields as DLL function parameters is no good idea anyway because there placement is dependent on the programming language and even on the compiler version use to build the original DLL.

Troubleshooting

If you experience trouble with the installation of DLLWrapper, please read the installation section of this manual carefully as it includes some necessary preconditions for a successful installation of the tool. Especially you should be sure that you've installed all required external tools before installing DLLWrapper itself.

If you encounter problems during the building of a wrapper DLL, please read the messages in the log tab of the GUI or on the console output carefully as they may give you some vital hints on what's going wrong. Typical sources for problems during the build of a wrapper DLL are wrong path entries made either during the installation or during the build or missing read and write access rights on selected folders.

If you experience trouble when trying to use a wrapper DLL, you should first check that the build wrapper DLL has actually been placed in a folder were it can be found by the host application (if not, the host application should issue a warning that for instance the file <target>_64.dll could not be located). If you're getting error message boxes from the wrapper DLL like "CoCreateInstance failed with error: 800...", you might have forgotten to register the COM objects using the <target>_reg.bat batch file (which requires admin rights and will fail without further notice, if you don't have them).

If you still have problems building or using a wrapper DLL, the tool provides some helpful diagnostic features that you can use to solve even complicated issues:

Verbose mode

The command line version of DLLWrapper has a command line switch (-V) to activate "verbose mode" in which all command invocations are echoed to the standard console output. If you're already using the command line version of DLLWrapper, simply add this option to your command line and hopefully you'll be able to find out which external tool is not running as it should and why. If you're using the GUI version of DLLWrapper, you'll find the command line information which is used by the GUI at the beginning of the messages displayed in the "Log" tab. You can copy and paste this command line setting to a console window and add the verbose mode command line switch there.

Logging code

You can enable the generation of logging code into the wrapper EXE. This code will not be active by default but can be activated by writing a registry key specific to the wrapper DLL. This can easily been done either for testing purposes right after building the wrapper DLL by using the respective GUI option or the command line option A or later maybe even on a different target machine by calling the generated <DLL-Name>_reg.bat batch file with the option /Logging to switch it on or without the option to switch logging of again.

If logging code has been generated and the logging feature has been activated every function invocation made through the generated wrapper EXE server to the original target DLL is logged with the values of all function parameters passed and the return value given back. This can be very helpful in order to find problems that occur when calling specific wrapped functions through the wrapper DLL from a 64 bit host application for instance. Of course this logging of every single function call can slow the DLL functions and possible the whole system down quite significantly so the logging feature should be deactivated again after finding a problem. Be aware that there is no size limit for the logging file right now which means that it will grow and grow until being deleted manually or until there is no space left on the storage device.

Service and Support

If the troubleshooting information above didn't help we'll provide 90 days e-mail support with 48 hours maximal response time for all customers that purchased a full license of DLLWrapper regarding installation and building issues after date of purchase - that means you can get quick, free-of-charge help if DLLWrapper is not installing or running as it should. Please understand that we can't provide support of any kind for the free-of-charge trail version of DLLWrapper.

However, we can NOT provide free support for wrapping a certain 32 bit DLL that you like to call from a 64 bit host of course, as there are trillions of possible use cases which are all individual and might lead to specific problems that have to solved individually, too.

For such cases (or if you simply don't like to do the wrapping by yourself) we provide a wrapping service for an additional fee which will build a wrapper package for a DLL provided by you according to your needs as long as you can provide the necessary header file and / or documentation to the DLL as well. If you're interested to make use of this wrapping service, please contact us under service@DLLWrapper.com and attach some files (like header files or documentation) that'll allow us to make a quote for wrapping the specific DLL interface that you need.

Please note however, that we typically won't be able to test the wrapping DLL in any way as we usually would need more than only the DLL to wrap but also the complete software and / or hardware environment in which the DLL is supposed to run for this. Therefore the test of the wrapping DLL is never part of our wrapping service but your responsibility - if you'll find bugs during your test (which can't be ruled out because no tests can't be run by us) and inform us accordingly we'll provide a fixed version of the wrapping DLL as part of the wrapping service until it is tested error-free by you. Finally, please be aware that it might not be possible to wrap all DLL functions transparently - see the chapter "Limitations" for some explanation. In such a case we'll try to provide the best possible work around...