FREE eBook

LEARNING Win32 API

Free unaffiliated eBook created from
Stack Overflow contributors.

#winapi

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: win32-api

It is an unofficial and free Win32 API ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Win32 API.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Win32 API

Remarks

WinAPI (also known as Win32; officially called the Microsoft Windows API) is an application programming interface written in C by Microsoft to allow access to Windows features. The main components of the WinAPI are:

- WinBase: The kernel functions, CreateFile, CreateProcess, etc
- WinUser: The GUI functions, CreateWindow, RegisterClass, etc
- WinGDI: The graphics functions, Ellipse, SelectObject, etc
- Common controls: Standard controls, list views, sliders, etc

See Also:

- Windows API index on MSDN.

Versions

Versions of the API are tied to the operating system version. MSDN documentation specifies the minimum supported operating system for each function in the API.

Examples

Hello World

Microsoft Windows applications are usually written as either a console application or a windowed application (there are other types such as services and plug-ins). The difference for the programmer is the difference in the interface for the main entry point for the application source provided by the programmer.

When a C or C++ application starts, the executable entry point used by the executable loader is the Runtime that is provided by the compiler. The executable loader reads in the executable, performs any fixup to the image needed, and then invokes the executable entry point which for a C or C++ program is the Runtime provided by the compiler.

The executable entry point invoked by the loader is not the main entry point provided by the application programmer but is instead the Runtime provided by the compiler and the linker which creates the executable. The Runtime sets up the environment for the application and then calls the main entry point provided by the programmer.

A Windows console application may have several slightly different interfaces for the main entry point provided by the programmer. The difference between these is whether the main entry point is the traditional `int main (int argc, char *argv[])` or if it is the Windows specific version of `int _tmain(int argc, _TCHAR* argv[])` which provides for wide characters in the application parameters.

If you generate a Windows Win32 console application project using Visual Studio, the source generated will be the Windows specific version.

A Windows window (GUI) application has a different interface for the main entry point provided by the programmer. This main entry point provided by the programmer has a more complex interface because the Runtime sets up a GUI environment and provides additional information along with the application parameters.

This example explains the Windows window (GUI) main entry point interface. To explore this topics you should have:

- an IDE with compiler (preferably Visual Studio)
- C knowledge

Create an empty Win32 windows (GUI, not console) project using the IDE. The project settings must be set for a window application (not a console application) in order for the linker to link with the correct Runtime. Create a `main.c` file adding it to the project and then type the following code:

```
#include <windows.h>

int APIENTRY WinMain(HINSTANCE hInst, HINSTANCE hInstPrev, PSTR cmdline, int cmdshow)
{
    return MessageBox(NULL, "hello, world", "caption", 0);
}
```

This is our Win32 "Hello, world" program. The first step is to include the windows header files. The main header for all of Windows is `windows.h`, but there are others.

The `WinMain` is different from a standard `int main()` used with a console application. There are more parameters used in the interface and more importantly the main entry point for a window application uses a calling convention different from standard C/C++.

The qualifier `APIENTRY` indicates the calling convention, which is the order in which arguments are pushed on the stack†. By default, the calling convention is the standard C convention indicated by `__cdecl`. However Microsoft uses a different type of calling convention, the PASCAL convention, for the Windows API functions which is indicated by the `__stdcall` qualifier. `APIENTRY` is a defined name for `__stdcall` in one of the header files included by `windows.h` (see also What is __stdcall?).

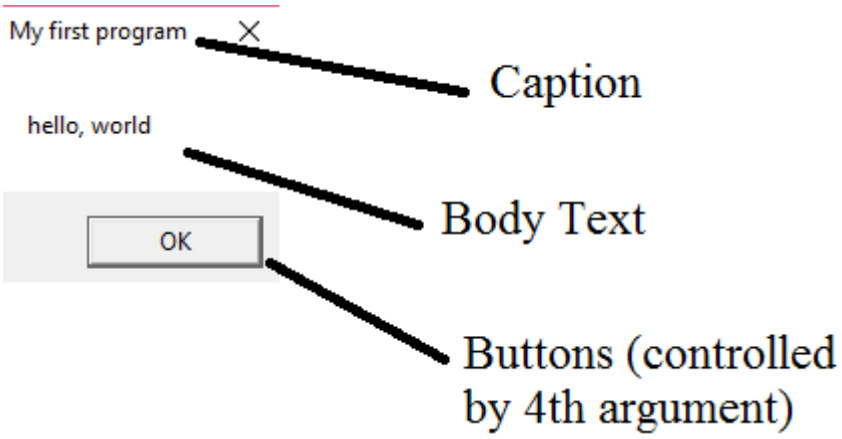
The next arguments to `WinMain` are as follows:

- hInst: The instance handle
- hInstPrev: The previous instance handle. No longer used.
- cmdline: Command line arguments (see Pass WinMain (or wWinMain) arguments to normal main)
- cmdshow: indicates if a window should be displayed.

We don't use any of these arguments yet.

Inside of `WinMain()`, is a call to `MessageBox()`, which displays a simple dialog with a message, a message box. The first argument is the handle to the owner window. Since we don't have our own

window yet, pass `NULL`. The second argument is the body text. The third argument is the caption, and the fourth argument contains the flags. When 0 is passed, a default message box is shown. The diagram below dissects the message box dialog.



Good links:

- Tutorial at winprog.org
- `MessageBox` function documentation at MSDN

†On 32 bit systems only. Other architectures have different calling conventions.

Read Getting started with Win32 API online: https://riptutorial.com/winapi/topic/1149/getting-started-with-win32-api

Chapter 2: Ansi- and Wide-character functions

Examples

Introduction

The Windows API documentation for functions taking one or more *string* as argument will usually look like this:

```
BOOL WINAPI CopyFile(
  _In_ LPCTSTR lpExistingFileName,
  _In_ LPCTSTR lpNewFileName,
  _In_ BOOL    bFailIfExists
);
```

The datatype for the two string parameters is made of several parts:

- LP = Long pointer
- C = const
- T = TCHAR
- STR = string

Now what does `TCHAR` mean? This depends on platform chosen for the compilation of program.

`CopyFile` itself is just a macro, defined something like this:

```
#ifdef UNICODE
#define CopyFile CopyFileW
#else
#define CopyFile CopyFileA
#endif
```

So there are actually two `CopyFile` functions and depending on compiler flags, the `CopyFile` macro will resolve to one or the other.

There core token, `TCHAR` is defined as:

```
#ifdef _UNICODE
typedef wchar_t TCHAR;
#else
typedef char TCHAR;
#endif
```

So again, depending on the compile flags, TCHAR is a "narrow" or a "wide" (2 bytes) character.

So when UNICODE is defined, `CopyFile` is defined to be `CopyFileW`, which will use 2-byte character arrays as their parameter, which are expected to be UTF-16 encoded.

If UNICODE isn't defined, `CopyFile` is defined to be `CopyFileA` which uses single-byte character arrays which are expected to be encoded in the default ANSI encoding of the current user.

There are two similar macros: `UNICODE` makes the Windows APIs expect wide strings and `_UNICODE` (with a leading underscore) which enables similar features in the C runtime library.

These defines allow us to write code that compiles in both ANSI and in Unicode-mode.

It is important to know that the ANSI encoding may be a single-byte encoding (i.e. latin-1) a multi-byte encoding (i.e. shift jis), although utf-8 is, unfortunately, not well supported.

This means that neither the ANSI, nor the Wide-character variant of these functions can be assumed to work with fixed width encodings.

Read Ansi- and Wide-character functions online: https://riptutorial.com/winapi/topic/2450/ansi--and-wide-character-functions

Chapter 3: Dealing with windows

Examples

Creating a window

```
#define UNICODE
#define _UNICODE
#include <windows.h>
#include <tchar.h>
const TCHAR CLSNAME[] = TEXT("helloworldWClass");
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp);

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, PTSTR cmdline,
                   int cmdshow)
{
    WNDCLASSEX wc = { };
    MSG msg;
    HWND hwnd;

    wc.cbSize        = sizeof (wc);
    wc.style         = 0;
    wc.lpfnWndProc   = winproc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInst;
    wc.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor (NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = CLSNAME;
    wc.hIconSm       = LoadIcon (NULL, IDI_APPLICATION);

    if (!RegisterClassEx(&wc)) {
        MessageBox(NULL, TEXT("Could not register window class"),
                   NULL, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindowEx(WS_EX_LEFT,
                          CLSNAME,
                          NULL,
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          NULL,
                          NULL,
                          hInst,
                          NULL);
    if (!hwnd) {
        MessageBox(NULL, TEXT("Could not create window"), NULL, MB_ICONERROR);
        return 0;
    }

    ShowWindow(hwnd, cmdshow);
```

```
    UpdateWindow(hwnd);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    return DefWindowProc(hwnd, wm, wp, lp);
}
```

The first thing one sees are the two macro definitions, `UNICODE` and `_UNICODE`. These macros cause our program to understand wide character strings (`wchar_t[n]`), not plain narrow strings(`char[n]`). As a result, all string literals must be wrapped in a `TEXT(` macro. The generic character type for Win32 strings is `TCHAR`, whose definition depends on whether or not `UNICODE` is defined. A new header is included: `<tchar.h>` contains the declaration of `TCHAR`.

A window consists of what is known as a *window class*. This describes information about a window that is to be shared between instances of it, like the icon, the cursor, and others. A window class is identified by a window class name, which is given in the `CLSNAME` global variable in this example. The first act of `WinMain` is to fill in the window class structure, `WNDCLASSEX wc`. The members are:

- cbSize: The size, in bytes, of the structure
- style: The window class styles. This is 0 for now.
- lpfnWndProc: This is one of the more important fields. It stores the address of the *window procedure*. The window procedure is a function that handles events for all windows that are instances of this window class.
- cbClsExtra: The number of extra bytes to allocate for the window class. For most situations, this member is 0.
- cbWndExtra: The number of extra bytes to allocate for each individual window. Do not confuse this with `cbClsExtra`, which is common to all instances. This is often 0.
- hInstance: The instance handle. Just assign the `hInst` argument in `WinMain` to this field.
- hIcon: The icon handle for the window class. `LoadIcon(NULL, IDI_APPLICATION)` loads the default application icon.
- hCursor: The cursor handle for the window class. `LoadCursor(NULL, IDC_ARROW)` loads the default cursor.
- hbrBackground: A handle to the background brush. `GetStockObject (WHITE_BRUSH)` gives a handle to a white brush. The return value must be cast because `GetStockObject` returns a generic object.
- lpszMenuName: The resource name of the menu bar to use. If no menu bar is needed, this field can be NULL.
- lpszClassName: The class name that identifies this window class structure. In this example, the `CLSNAME` global variable stores the window class name.
- hIconSm: A handle to the small class icon.

After this structure is initialized, the `RegisterClassEx` function is called. This causes the window class to be registered with Windows, making it known to the application. It returns 0 on failure.

Now that the window class has been registered, we can display the window using `CreateWindowEx`. The arguments are:

- stylesex: The extended window styles. The default value is WS_EX_LEFT.
- clsname: The class name
- cap: The window title, or caption. In this case, it is the caption that is displayed in a window's title bar.
- styles: The window styles. If you want to create a top-level (parent) window like this one, the flag to pass in is WS_OVERLAPPEDWINDOW.
- x: The x-coordinate of the upper-left corner of the window.
- y: The y-coordinate of the upper-left corner of the window
- cx: The width of the window
- cy: The height of the window
- hwndParent: The handle to the parent window. Since this window is in itself a parent window, this argument is NULL.
- hMenuOrID: If the window being created is a parent window, then this argument is a handle to the window menu. Do not confuse this with the class menu, which is `WNDCLASSEX::lpszClassName`. The class menu is common to all instances of windows with the same class name. This argument, however, is specific for just this instance. If the window being created is a child window, then this is the ID of the child window. In this case, we are creating a parent window with no menu, so NULL is passed.
- hInst: The handle to the instance of the application.
- etc: The extra information that is passed to the window's window procedure. If no extra information is to be transmitted, pass NULL.

If `x` or `y` or `cx` or `cy` is `CW_USEDEFAULT`, then that argument's value will be determined by Windows. That is what is done in this example.

`CreateWindowEx` returns the handle to the newly created window. If window creation failed, it returned `NULL`.

We then show the window by calling `ShowWindow`. The first argument for this function is the handle to the window. The second argument is the show style, which indicates how the window is to be displayed. Most applications just pass the `cmdshow` argument passed in `WinMain`. After the window is shown, it must be updated by a call to `UpdateWindow`. It causes an update message to be sent to the window. We will learn what this means in another tutorial.

Now comes the heart of the application: The message pump. It pumps messages sent to this application by the operating system, and dispatches the messages to the window procedure. The `GetMessage` call returns non-zero until the application receieves a messages that causes it to quit, in which case it returns 0. The only argument that concerns us is the pointer to an `MSG` structure that will be filled in with information about the message. The other arguments are all 0.

Inside the message loop, `TranslateMessage` translates virtual-key messages into character messages. The meaning of this, again, is unimportant to us. It takes a pointer to an `MSG` structure. The call directly following it, `DispatchMessage`, dispatches the message pointed to by its argument to the window's window procedure. The last thing `WinMain` must do is return a status code. The `wParam` member of the `MSG` structure contains this return value, so it is returned.

But that's just for the `WinMain` function. The other function is `winproc`, the window procedure. It will handle messages for the window that are sent to it by Windows. The signature for `winproc` is:

- hwnd: A handle to the window whose messages are being processed.
- wm: The window message identifier
- wp: One of the message information arguments. This depends on the `wm` argument
- lp: One of the message information arguments. This depends on the `wm` argument. This argument is usually used to transmit pointers or handles

In this simple program, we do not handle any messages ourselves. But that doesn't mean Windows doesn't either. This is why one must call `DefWindowProc`, which contains default window handling code. This function must be called at the end of every window procedure.

What is a handle?

A *handle* is a data type that represents a unique object. They are pointers, but to secret data structures maintained by the operating system. The details of these structures need not concern us. All a user needs to do is simply create/retreive a handle using an API call, and pass it around to other API calls taking that type of handle. The only type of handle we used was the `HWND` returned by `CreateWindowEx`.

Constants

In this example, we encounter a handful of constants, which are in all-caps and begin with a 2 or 3 letter prefix. (The Windows types are also in all-caps)

- IDI_APPLICATION: The resource name containing the default application icon. This is used with either `LoadIcon` or `LoadImage` (LoadIcon in this example).
- IDC_ARROW: The resource name countaining the default application cursor. This is used with either `LoadIcon` or `LoadImage` (LoadIcon in this example).
- WHITE_BRUSH: The name of a stock object. This stock object is the white brush.
- MB_ICONERROR: A flag used with `MessageBox` to display an error icon.
- WS_EX_LEFT: The default extended window style. This causes the window to have left-aligned properties.
- WS_OVERLAPPEDWINDOW: A window style indicating that the window should be a parent window with a title bar, size box, and others elements typical of top-level windows.
- CW_USEDEFAULT: Used with `CreateWindowEx`'s `x`, `y`, `cx`, or `cy` arguments. Causes Windows to choose a valid value for the argument for which `CW_USEDEFAULT` was passed.

Windows Types

When programming for Windows, you will have to get used to the Win32 types, which are aliases for builtin types. These types are in all caps. The alias types used in this program are:

- TCHAR: The generic character type. If `UNICODE` is defined, this is a `wchar_t`. Otheriwse, it is a `char`.
- UINT: An unsigned integer. Used to represent the message identifier in window procedures,

and other purposes.

- WPARAM: In Win16, this was a WORD argument (hence the `W` prefix). With the introduction of Win32, however, this is now a `UINT_PTR`. This illustrates the point of these Windows aliases; they are there to protect programs from change.
- LPARAM: This is a `LONG` argument (`LONG_PTR` in Win64).
- PTSTR: The `P` means pointer. The `T` means generic character, and the `STR` means string. Thus, this is a pointer to a `TCHAR` string. Other string types include:
    - LPTSTR: Same as `PTSTR`
    - LPCTSTR: Means `const TCHAR *`
    - PCTSTR: Same as `LPCTSTR`
    - LPWSTR: Wide string (`wchar_t *`)
    - LPCWSTR: Means `const wchar_t *`
    - PWSTR: Same as `LPWSTR`
    - and much more As you can see, the Win32 types can be a hassle to understand, especially with so many synonymous types, which is an artifact of Win16.
- LRESULT: This type is used to represent the return value of window procedures. It is usually a LONG (hence the `L`).

Read Dealing with windows online: https://riptutorial.com/winapi/topic/2782/dealing-with-windows

Chapter 4: Error reporting and handling

Remarks

Each thread will have its own last error code. The Windows API will set the last error code on the calling thread.

You should always call the `GetLastError` function immediately after checking a Windows API function's return value.

The majority of Windows API functions set the last error code when they fail. Some will also set the last error code when they succeed. There are a number of functions that do not set the last error code. Always refer to the Windows API function's documentation.

It is unsafe to use `FORMAT_MESSAGE_FROM_SYSTEM` without `FORMAT_MESSAGE_IGNORE_INSERTS` when using the `FormatMessage` function to get a description of an error code.

Examples

Introduction

The Windows API is provided by means of a C-callable interface. Success or failure of an API call is reported strictly through return values. Exceptions aren't part of the documented contract (although some API **implementations** can raise SEH exceptions, e.g. when passing a read-only *lpCommandLine* argument to CreateProcess).

Error reporting roughly falls into one of four categories:

- Return value only
- Return value with additional information on failure
- Return value with additional information on failure and success
- `HRESULT` return value

The documentation for each API call explicitly calls out, how errors are reported. Always consult the documentation.

Error reported by return value only

Some API calls return a single failure/success flag, without any additional information (e.g. GetObject):

```
if ( GetObjectW( obj, 0, NULL ) == 0 ) {
    // Failure: no additional information available.
}
```

Error reported with additional information on failure

In addition to a failure/success return value, some API calls also set the last error on failure (e.g. CreateWindow). The documentation usually contains the following standard wording for this case:

> If the function succeeds, the return value is *<API-specific success value>*.
> If the function fails, the return value is *<API-specific error value>*. To get extended error information, call **GetLastError**.

```
if ( CreateWindowW( ... ) == NULL ) {
    // Failure: get additional information.
    DWORD dwError = GetLastError();
} else {
    // Success: must not call GetLastError.
}
```

It is vital that you call `GetLastError()` IMMEDIATELY. The last error code can be overwritten by any other function, so if there's an extra function call between the function that failed and the call to `GetLastError()`, the return from `GetLastError()` will no longer be reliable. Take extra caution when dealing with C++ constructors.

Once you get an error code, you will need to interpret it. You can get a comprehensive list of error codes on MSDN, at the System Error Codes (Windows) page. Alternatively, you can look in your system header files; the file with all the error code constants is `winerror.h`. (If you have Microsoft's official SDK for Windows 8 or newer, this is in the `shared` subfolder of the include folder.)

Notes on calling `GetLastError()` in other programming languages

.net languages (C#, VB, etc.)

With .net, you **should not** P/Invoke to `GetLastError()` directly. This is because the .net runtime will make other Windows API calls on the same thread behind your back. For instance, the garbage collector might call `VirtualFree()` if it finds enough memory that it is no longer using, *and this can happen between your intended function call and your call to `GetLastError()`*.

Instead, .net provides the `Marshal.GetLastWin32Error()` function, which will retrieve the last error from the last P/Invoke call that you yourself made. Use this instead of calling `GetLastError()` directly.

(.net does not seem to stop you from importing `GetLastError()` anyway; I'm not sure why.)

Go

The various facilities provided by Go for calling DLL functions (which reside in both package `syscall` and package `golang.org/x/sys/windows`) return three values: `r1`, `r2`, and `err`. `r2` is never used; you can use the blank identifier there. `r1` is the function's return value. `err` is the result of calling `GetLastError()` but converted into a type that implements `error`, so you can pass it up to calling functions to handle.

Because Go does not know when to call `GetLastError()` and when not to, it will **always** return a

non-`nil` error. Therefore, the typical Go error-handling idiom

```
r1, _, err := syscall.Syscall12(CreateWindowW.Addr(), ...)
if err != nil {
    // handle err
}
// use r1
```

will not work. Instead, you must check `r1`, exactly as you would in C, and only use `err` if *that* indicates the function returned an error:

```
r1, _, err := syscall.Syscall12(CreateWindowW.Addr(), ...)
if r1 == 0 {
    // handle err
}
// use r1
```

Error reported with additional information on failure and success

Some API calls can succeed or fail in more than one way. The APIs commonly return additional information for both successful invocations as well as errors (e.g. CreateMutex).

```
if ( CreateMutexW( NULL, TRUE, L"Global\\MyNamedMutex" ) == NULL ) {
    // Failure: get additional information.
    DWORD dwError = GetLastError();
} else {
    // Success: Determine which mutex was returned.
    if ( GetLastError() == ERROR_ALREADY_EXISTS ) {
        // Existing mutex object returned.
    } else {
        // Newly created mutex object returned.
    }
}
```

Error reported as HRESULT value

HRESULTs are numeric 32-bit values, where bits or bit ranges encode well-defined information. The MSB is a failure/success flag, with the remaining bits storing additional information. Failure or success can be determined using the FAILED or SUCCEEDED macros. `HRESULT`s are commonly used with COM, but appear in non-COM implementations as well (e.g. StringCchPrintf).

```
const size_t cchBuf = 5;
wchar_t buffer[cchBuf] = { 0 };
HRESULT hr = StringCchPrintfW( buffer, cchBuf, L"%s", L"Hello, world!" );
if ( FAILED( hr ) ) {
    // Failure: Determine specific reason.
    switch ( hr ) {
    case STRSAFE_E_INSUFFICIENT_BUFFER:
        // Buffer too small; increase buffer and retry.
        ...
    case STRSAFE_E_INVALID_PARAMETER:
        // Invalid parameter; implement custom error handling (e.g. logging).
        ...
```

```
    default:
        // Some other error code; implement custom error handling (e.g. logging).
        ...
    }
}
```

Converting an error code into a message string

`GetLastError` returns a numerical error code. To obtain a descriptive error message (*e.g.*, to display to a user), you can call `FormatMessage`:

```
// This functions fills a caller-defined character buffer (pBuffer)
// of max length (cchBufferLength) with the human-readable error message
// for a Win32 error code (dwErrorCode).
//
// Returns TRUE if successful, or FALSE otherwise.
// If successful, pBuffer is guaranteed to be NUL-terminated.
// On failure, the contents of pBuffer are undefined.
BOOL GetErrorMessage(DWORD dwErrorCode, LPTSTR pBuffer, DWORD cchBufferLength)
{
    if (cchBufferLength == 0)
    {
        return FALSE;
    }

    DWORD cchMsg = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                                 NULL,  /* (not used with FORMAT_MESSAGE_FROM_SYSTEM) */
                                 dwErrorCode,
                                 MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                                 pBuffer,
                                 cchBufferLength,
                                 NULL);
    return (cchMsg > 0);
}
```

In **C++**, you can simplify the interface considerably by using the `std::string` class:

```
#include <Windows.h>
#include <exception>
#include <stdexcept>
#include <memory>
#include <string>
typedef std::basic_string<TCHAR> String;

String GetErrorMessage(DWORD dwErrorCode)
{
    LPTSTR psz = NULL;
    const DWORD cchMsg = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM
                                       | FORMAT_MESSAGE_IGNORE_INSERTS
                                       | FORMAT_MESSAGE_ALLOCATE_BUFFER,
                                     NULL, // (not used with FORMAT_MESSAGE_FROM_SYSTEM)
                                     dwErrorCode,
                                     MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                                     reinterpret_cast<LPTSTR>(&psz),
                                     0,
```

```
                                        NULL);
    if (cchMsg > 0)
    {
        // Assign buffer to smart pointer with custom deleter so that memory gets released
        // in case String's c'tor throws an exception.
        auto deleter = [](void* p) { ::HeapFree(::GetProcessHeap(), 0, p); };
        std::unique_ptr<TCHAR, decltype(deleter)> ptrBuffer(psz, deleter);
        return String(ptrBuffer.get(), cchMsg);
    }
    else
    {
        throw std::runtime_error("Failed to retrieve error message string.");
    }
}
```

NOTE: These functions also work for `HRESULT` values. Just change the first parameter from `DWORD dwErrorCode` to `HRESULT hResult`. The rest of the code can remain unchanged.

Read Error reporting and handling online: https://riptutorial.com/winapi/topic/2573/error-reporting-and-handling

Chapter 5: File Management

Examples

Create a file and write to it

This example creates a new file named "NewFile.txt", then writes "Hello World!" to its body. If the file already exists, `CreateFile` will fail and no data will be written. See the `dwCreationDisposition` parameter in the CreateFile documentation if you don't want the function to fail if the file already exists.

```
#include <Windows.h>
#include <string>

int main()
{
   // Open a handle to the file
   HANDLE hFile = CreateFile(
      L"C:\\NewFile.txt",     // Filename
      GENERIC_WRITE,          // Desired access
      FILE_SHARE_READ,        // Share mode
      NULL,                   // Security attributes
      CREATE_NEW,             // Creates a new file, only if it doesn't already exist
      FILE_ATTRIBUTE_NORMAL,  // Flags and attributes
      NULL);                  // Template file handle

   if (hFile == INVALID_HANDLE_VALUE)
   {
      // Failed to open/create file
      return 2;
   }

   // Write data to the file
   std::string strText = "Hello World!"; // For C use LPSTR (char*) or LPWSTR (wchar_t*)
   DWORD bytesWritten;
   WriteFile(
      hFile,            // Handle to the file
      strText.c_str(),  // Buffer to write
      strText.size(),   // Buffer size
      &bytesWritten,    // Bytes written
      nullptr);         // Overlapped

   // Close the handle once we don't need it.
   CloseHandle(hFile);
}
```

API Reference:

- MSDN `CreateFile`
- MSDN `WriteFile`

Read File Management online: https://riptutorial.com/winapi/topic/1765/file-management

Chapter 6: Process and Thread Management

Examples

Create a process and check its exit code

This example starts Notepad, waits for it to be closed, then gets its exit code.

```
#include <Windows.h>

int main()
{
   STARTUPINFOW si = { 0 };
   si.cb = sizeof(si);
   PROCESS_INFORMATION pi = { 0 };

   // Create the child process
   BOOL success = CreateProcessW(
      L"C:\\Windows\\system32\\notepad.exe",  // Path to executable
      NULL,                                  // Command line arguments
      NULL,                                  // Process attributes
      NULL,                                  // Thread attributes
      FALSE,                                 // Inherit handles
      0,                                     // Creation flags
      NULL,                                  // Environment
      NULL,                                  // Working directory
      &si,                                   // Startup info
      &pi);                                  // Process information

   if (success)
   {
      // Wait for the process to exit
      WaitForSingleObject(pi.hProcess, INFINITE);

      // Process has exited - check its exit code
      DWORD exitCode;
      GetExitCodeProcess(pi.hProcess, &exitCode);

      // At this point exitCode is set to the process' exit code

      // Handles must be closed when they are no longer needed
      CloseHandle(pi.hThread);
      CloseHandle(pi.hProcess);
   }
}
```

References (MSDN):

- CreateProcess
- WaitForSingleObject
- GetExitCodeProcess
- CloseHandle

Create a new thread

```
#include <Windows.h>

DWORD WINAPI DoStuff(LPVOID lpParameter)
{
    // The new thread will start here
    return 0;
}

int main()
{
    // Create a new thread which will start at the DoStuff function
    HANDLE hThread = CreateThread(
        NULL,     // Thread attributes
        0,        // Stack size (0 = use default)
        DoStuff, // Thread start address
        NULL,     // Parameter to pass to the thread
        0,        // Creation flags
        NULL);    // Thread id
    if (hThread == NULL)
    {
        // Thread creation failed.
        // More details can be retrieved by calling GetLastError()
        return 1;
    }

    // Wait for thread to finish execution
    WaitForSingleObject(hThread, INFINITE);

    // Thread handle must be closed when no longer needed
    CloseHandle(hThread);

    return 0;
}
```

Note that the CRT also provides the `_beginthread` and `_beginthreadex` APIs for creating threads, which are not shown in this example. The following link discusses the differences between these APIs and the `CreateThread` API.

References (MSDN):

- `CreateThread`
- `WaitForSingleObject`
- `CloseHandle`
- `_beginthread`, `_beginthreadex`

Read Process and Thread Management online: https://riptutorial.com/winapi/topic/1756/process-and-thread-management

Chapter 7: Utilizing MSDN Documentation

Introduction

The Windows API is vast, and contains a lot of features. The size of the API is such that no one can know all of it. While there are many resources like StackOverflow, there is no substitute for the official documentation.

Remarks

Examples of Documentation:

- **Topic Overview:** Desktop Window Manager Performance Considerations and Best Practices
- **Samples:** Customize an Iconic Thumbnail and a Live Preview Bitmap
- **Functions:** DwmSetIconicThumbnail function

Examples

Types of Documentation Available

The MSDN library contains several different types of documentation which can be used for implementing features.

- **Topic Overviews** These are broad overviews of topics intended to provide a general understanding of an API. These overviews also often outline best practices, and implementation strategies.
- **Samples** Demonstrate the use of particular APIs. These are generally highly simplified, don't necessarily do error checking, and typically don't use frameworks like MFC or ATL. They provide a starting point for using features.
- **Reference** Details all of the elements of each API. This includes constants/enumerations, interfaces, functions and classes.

Note: Many Microsoft employees also maintain blogs, like Raymond Chen's The Old New Thing that can supplement the documentation, but these blogs are not a substitute for the documentation.

Finding Documentation for a Feature

Finding documentation for a feature is often as simple as a search using a good search engine. If that fails, or if unsure about specific terms, the Windows API Index can help locate specific features. Documentation for methods, interfaces, enumerations and constants can usually be found by searching for the name using a search engine. Additionally, the Windows Dev Center can provide a valuable starting point.

Using Function Documentation

The documentation for a function is broken down into several sections:

Overview

Describes what the function is used for. This section will also show information about whether the function is depreciated, or may be unavailable in future versions.

Syntax

Shows the declaration of the function from the appropriate source header. It is a quick reference to the function's signature.

Parameters

Explains each of the parameters, whether the parameter is input or output, and other important considerations.

Return Value

This section explains the result of the function call, including how to detect errors, and what additional information is available. (For example, this section will state explicitly if `GetLastError` will provide additional error handling information.)

Remarks

Covers any additional information required to use the function, such as, information about supporting functions, obtaining appropriate handles, and disposal of resources.

Examples

If this section is available, it has an example of the appropriate use of the function to use as a starting point for implementation.

Requirements

Gives important information about prerequisites for calling the function. This information includes:

- **Minimum Supported Client/Server** First version of the operating system (supported by Microsoft) to provide the function.

  (Note that this field is notoriously misleading. Often, functions are supported in an earlier version of the operating system, but this field only shows the earliest version that is *currently supported by Microsoft*. For example, the `CreateWindow` function has been supported since Windows 1.0, but the documentation only shows that it has been supported since Windows 2000. The online version of the MSDN documentation does not indicate that *any* function was supported in a version of Windows prior to 2000, even though many were. For legacy development, you will need to consult an older version of the SDK documentation, such as might have been shipped on an MSDN CD-ROM. Or, just look in the header files.)
- **Header** The SDK header to `#include` that contains the function declaration. If the function isn't available in a header, this will show information about the procedure to call the function (usually calling `GetProcAddress` to do run-time dynamic linking).
- **Library** The library file to pass to the linker to resolve the exported functions.
- **DLL** The file (as shipped with the operating system) that contains the exported function.
- **End of Client/Server Support** The last version of Windows to officially support the API.
- **Unicode and ANSI names** For string functions that have both Unicode and ANSI variants, this lists the actual exported names for the two functions. This is usually just the function name with a `W` or `A` suffix (respectively).

Read Utilizing MSDN Documentation online: https://riptutorial.com/winapi/topic/8999/utilizing-msdn-documentation

Chapter 8: Window messages

Syntax

- #include <windows.h>
- BOOL WINAPI DestroyWindow(HWND hwnd);
- VOID WINAPI PostQuitMessage(int exitcode);
- BOOL WINAPI MoveWindow(HWND hwnd, int x, int y, int cx, int cy, BOOL bRepaint);

Examples

WM_CREATE

A WM_CREATE message is sent to your window procedure during the window's `CreateWindowEx` call. The `lp` argument contains a pointer to a `CREATESTRUCT` which contains the arguments passed to `CreateWindowEx`. If an application returns 0 from WM_CREATE, the window is created. If an application returns -1, creation is canceled.

```
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    switch (wm) {
        case WM_CREATE:
            CREATESTRUCT *cs = (CREATESTRUCT *) lp;
            if (MessageBox(hwnd,
                "Do you want to continue creating the window?", "", MB_YESNO)
                                        == IDYES) {
                /* create window controls */
                return 0;
            }
            /* cancel creation */
            return -1;
    }
    return DefWindowProc(hwnd, wm, wp, lp);
}
```

WM_DESTROY

This message is sent to your window procedure when a window is being destroyed. It is sent after the window is removed from the screen. Most applications free any resources, like memory or handles, obtained in WM_CREATE. If you handle this message, return 0.

```
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    static char *text;
    switch (wm) {
        case WM_CREATE:
            text = malloc(256);
            /* use the allocated memory */
            return 0;
        case WM_CLOSE:
```

```
             switch (MessageBox(hwnd, "Save changes?", "", MB_YESNOCANCEL)) {
                 case IDYES:
                      savedoc();
                                       /* fall through */
                 case IDNO:
                      DestroyWindow(hwnd);
                      break;
             }
             return 0;
        case WM_DESTROY:
            /* free the memory */
            free(text);
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, wm, wp, lp);
}
```

WM_CLOSE

Sent when an application's close button is clicked. Do not confuse this with `WM_DESTROY` which is sent when a window will be destroyed. The main difference lies in the fact that closing may be canceled in WM_CLOSE (think of Microsoft Word asking to save your changes), versus that destroying is when the window has already been closed (think of Microsoft Word freeing memory).

```
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    static char *text;
    switch (wm) {
        case WM_CREATE:
            text = malloc(256);
            /* use the allocated memory */
            return 0;
        case WM_CLOSE:
             switch (MessageBox(hwnd, "Save changes?", "", MB_YESNOCANCEL)) {
                 case IDYES:
                      savedoc();
                                       /* fall through */
                 case IDNO:
                      DestroyWindow(hwnd);
                      break;
             }
             return 0;
        case WM_DESTROY:
            /* free the memory */
            free(text);
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, wm, wp, lp);
}
```

WM_SIZE

This message is sent to the window's window procedure after it's size has changed. The most common reason for handling this message is to adjust the position of any child windows. For

example, in Notepad, when the window is resized the child window (edit control) is also resized. Return 0 if you handle this message.

Argument	Value
wp	One of the window sizing constants.
lp	LOWORD(lp) is the new width of the client area HIWORD(lp) is the new height of the client area.

```
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    switch (wm) {
        case WM_SIZE:
            /* hwndEdit is the handle of the edit control window */
            MoveWindow(hwndEdit, 0, 0, LOWORD(lp), HIWORD(lp), TRUE);
            return 0;
   }
   return DefWindowProc(hwnd, wm, wp, lp);
}
```

WM_COMMAND

Sent to a window procedure when:

- the user selects an item from a menu
- a control sends a notification to its parent window
- an accelerator keystroke is translated

Message Source	HIWORD(wp)	LOWORD(wp)	lp
Menu	0	Menu ID (IDM_*)	0
Accelerator	1	Accel ID (IDM_*)	0
Control	notification code	Control id	HWND of control window

For example, in Notepad, when a user clicks "File->Open" a dialog box is displayed to allow the user to open a file. Menu items are processed in the window procedure's WM_CREATE message like this:

```
LRESULT CALLBACK winproc(HWND hwnd, UINT wm, WPARAM wp, LPARAM lp)
{
    switch (wm) {
        case WM_COMMAND:
            switch (LOWORD(wp) {
                case ID_FILE_OPEN:
                    /* show file open dialog */
                    break;
                case ID_FILE_NEW:
                    /* create new instance */
```

```
                    break;
            }
            return 0;
    }
    return DefWindowProc(hwnd, wm, wp, lp);
}
```

Read Window messages online: https://riptutorial.com/winapi/topic/2449/window-messages

Chapter 9: Windows Services

Examples

Check if a service is installed

This example show how you can check if a service already exists (*i.e.*, is installed on the machine) or not. This code requires only the lowest privileges necessary, so each process can perform the check, no matter what level of security it is running at.

```
#define UNICODE
#define _UNICODE
#include <Windows.h>
#include <string>
#include <iostream>

enum Result
{
   unknown,
   serviceManager_AccessDenied,
   serviceManager_DatabaseDoesNotExist,
   service_AccessDenied,
   service_InvalidServiceManagerHandle,
   service_InvalidServiceName,
   service_DoesNotExist,
   service_Exist
};

Result ServiceExists(const std::wstring &serviceName)
{
   Result r = unknown;

   // Get a handle to the SCM database
   SC_HANDLE manager = OpenSCManager(NULL, SERVICES_ACTIVE_DATABASE, GENERIC_READ);

   if (manager == NULL)
   {
      DWORD lastError = GetLastError();

      // At this point, we can return directly because no handles need to be closed.
      if (lastError == ERROR_ACCESS_DENIED)
         return serviceManager_AccessDenied;
      else if (lastError == ERROR_DATABASE_DOES_NOT_EXIST)
         return serviceManager_DatabaseDoesNotExist;
      else
         return unknown;
   }

   SC_HANDLE service = OpenService(manager, serviceName.c_str(), GENERIC_READ);

   if (service == NULL)
   {
      DWORD error = GetLastError();

      if (error == ERROR_ACCESS_DENIED)
         r = service_AccessDenied;
```

```
      else if (error == ERROR_INVALID_HANDLE)
         r = service_InvalidServiceManagerHandle;
      else if (error == ERROR_INVALID_NAME)
         r = service_InvalidServiceName;
      else if (error == ERROR_SERVICE_DOES_NOT_EXIST)
         r = service_DoesNotExist;
      else
         r = unknown;
   }
   else
      r = service_Exist;

   if (service != NULL)
      CloseServiceHandle(service);

   if (manager != NULL)
      CloseServiceHandle(manager);

   return r;
}

int main()
{
   std::wstring serviceName = L"MSSQL$SQLEXPRESS";  // name of the service to check
   Result result = ServiceExists(serviceName);
   if (result == service_Exist)
      std::wcout << L"The service '" << serviceName << "' exists." << std::endl;
   else if (result == service_DoesNotExist)
      std::wcout << L"The service '" << serviceName << "' does not exist." << std::endl;
   else
      std::wcout << L"An error has occurred, and it could not be determined whether the
service '" << serviceName << "' exists or not." << std::endl;
}
```

API Reference:

- MSDN OpenSCManager
- MSDN OpenService
- MSDN CloseServiceHandle

Read Windows Services online: https://riptutorial.com/winapi/topic/2256/windows-services

Chapter 10: Windows Subclassing

Introduction

Window subclassing is a way to hook up into standard window procedure and to modify or extend its default behavior. An application subclasses a window by replacing the the window's original window procedure with a new window procedure. This new window procedure receives any messages sent or posted to the window.

Syntax

- BOOL SetWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId, DWORD_PTR RefData);
- BOOL RemoveWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId);
- BOOL GetWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId, DORD_PTR* RefData);
- LRESULT DefSubclassProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);

Parameters

Parameter	Detail
hWnd	The handle of the window to subclass.
SubclassProc	The subclass callback procedure.
SubclassId	User specified ID to identify the subclass, together with the subclass procedure uniquely identifies a subclass. It can simply be an arbitrary consecutive number.
RefData	User specified data. The meaning is determined by the application. It is passed to the subclass callback in unmodified way. It could be an object pointer to a class instance for example.

Remarks

MSDN Documentation

- About Windows Procedures
- Subclassing Controls

Examples

Subclassing windows button control within C++ class

This example shows how to manipulate button ideal size by specifying a fixed size.

```
class ButtonSubclass {
public:

    ButtonSubclass(HWND hWndButton) {
        SetWindowSubclass(hWndButton, MyButtonSubclassProc, 1, (DWORD_PTR) this);
    }
    ~ButtonSuclass() {
        RemoveWindowSubclass(hWndButton, MyButtonSubclassProc, 1, (DWORD_PTR) this);
    }

protected:

    static LRESULT CALLBACK MyButtonSubclassProc(
           HWND hWnd, UINT Msg, WPARAM w, LPARAM l, DWORD_PTR RefData) {

        ButtonSubclass* o = reinterpret_cast<ButtonSubclass*>(RefData);

        if (Msg == BCM_GETIDEALSIZE) {
            reinterpret_cast<SIZE*>(lParam)->cx = 100;
            reinterpret_cast<SIZE*>(lParam)->cy = 100;
            return TRUE;
        }
        return DefSubclassProc(hWnd, Msg, w, l);
    }
}
```

Installing and removing subclass procedure

The following methods installs or removes the subclass callback. The combination of `SubclassId` and `SubclassProc` uniquely identifies a subclass. There is no reference counting, calling `SetWindowSubclass` multiple times with different `RefData` only updates that value but will not causes the subclass callback to be called multiple times.

```
BOOL SetWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId, DWORD_PTR
RefData);
BOOL RemoveWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId);
```

To retrieve the reference data that was passed in the last `SetWindowSubclass`call, one can use the `GetWindowSubclass` method.

```
BOOL GetWindowSubclass(HWND hWnd, SUBCLASSPROC SubclassProc, UINT_PTR SubclassId, DORD_PTR*
RefData);
```

Parameter	Detail
hWnd	The handle of the window to subclass.

Parameter	Detail
SubclassProc	The subclass callback procedure.
SubclassId	User specified ID to identify the subclass, together with the subclass procedure uniquely identifies a subclass. It can simply be an arbitrary consecutive number.
RefData	User specified data. The meaning is determined by the application. It is passed to the subclass callback in unmodified way. It could be an object pointer to a class instance for example.

The subclass callback is responsible to call the next handler in window's subclass chain. `DefSubclassProc` calls the next handler in window's subclass chain. The last handler calls the original window procedure. It should be called in any subclassing callback procedure unless the message is completely handled by the application.

```
LRESULT DefSubclassProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Parameter	Detail
hWnd	Window handle where the message originates from
Msg	Window message
wParam	WPARAM argument, this value depends on specific window message
lParam	LPARAM argument, this value depends on specific window message

SUBCLASSPROC

It is similar to `WINDOWPROC` callback but contains an additional argument `RefData`.

```
typedef LRESULT (CALLBACK *SUBCLASSPROC)(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam,
    UINT_PTR SubclassId,
    DWORD_PTR RefData
);
```

Handling common controls notification messages within C++ class

```
class MyToolbarControl {
public:
    MyToolbarControl(HWND hWndToolbar, HWND hWndNotifyParent = nullptr) : _Handle(hWndToolbar)
{
        if (hWndNotifyParent == nullptr) {
            hWndNotifyParent = GetAncestor(hWndToolbar, GA_ROOTOWNER);
```

```
        }
        SetWindowSubclass(
            hWndNotifyParent , SubclassWindowProc, reinterpret_cast<UINT_PTR>(this),
reinterpret_cast<DWORD_PTR>(this)
        );
    }
    ~MyToolbarControl() {
        RemoveWindowSubclass(
            hWndNotifyParent , SubclassWindowProc, reinterpret_cast<UINT_PTR>(this),
reinterpret_cast<DWORD_PTR>(this)
        );
    }

protected:
    HWND _Handle;

    static LRESULT CALLBACK SubclassWindowProc(
        HWND hWnd, UINT Msg, WPARAM w, LPARAM l, UINT_PTR SubclassId, DWORD_PTR RefData) {
        MyToolbarControl * w = reinterpret_cast<MyToolbarControl *>(RefData);
        if (Msg == WM_NOTIFY) {
            NMHDR* h = reinterpret_cast<NMHDR*>(l);
            if (h->hwndFrom == w->_Handle) {
                // Handle notification message here...
            }
        }
        return DefSubclassProc(hWnd, Msg, w, l);
    }
};
```

Read Windows Subclassing online: https://riptutorial.com/winapi/topic/9399/windows-subclassing

Credits

S. No	Chapters	Contributors
1	Getting started with Win32 API	Community, IInspectable, Richard Chambers, stackptr, Stuart, theB
2	Ansi- and Wide-character functions	Adrian McCarthy, Ajay, IInspectable, Tannin
3	Dealing with windows	stackptr
4	Error reporting and handling	Ajay, andlabs, camelCase, Cody Gray, IInspectable, stackptr
5	File Management	Adi Lester, Ajay, theB
6	Process and Thread Management	Adi Lester, Ajay, IInspectable, theB
7	Utilizing MSDN Documentation	Cody Gray, theB
8	Window messages	stackptr
9	Windows Services	Cody Gray, Mr. Gray
10	Windows Subclassing	bkausbk