

maXbox



# maXbox Starter 37

## Start API Coding

### 1.1 A Programming Interface

Today we step through the API Coding of Windows.

Although, you probably wondered how they get those nice and fancy graphical user interfaces (GUI) in windows programs that don't have a black screen. Well, these GUI programs are called Win32 API programs. Learn how to make API calls with a black screen and other GUI objects in Windows 32 programming.

Lets start with a first function of a API function call:

if FConnect then

```
Result:= WNetConnectionDialog(GetForegroundWindow,  
                                RESOURCETYPE_DISK) = NO_ERROR
```

The `WNetConnectionDialog` function starts a general browsing dialog box for connecting to network resources. The function requires a handle to the owner window for the visual dialog box.

Syntax in C++

```
DWORD WNetConnectionDialog(  
    _In_ HWND hwnd,  
    _In_ DWORD dwType);
```

You do have 2 parameters: A handle to the owner window for the dialog box and `dwType` as a resource type to allow connections to.

Also the return value as `DWORD` has to be considered:

If the function succeeds, the return value is `NO_ERROR`. If the user cancels the dialog box, the function returns `-1` as the boolean result above.

If the function fails, the return value is a system error code, such as one of the following values:

`ERROR_EXTENDED_ERROR`

A network-specific error occurred. To obtain a description of the error, call the `WNetGetLastError` function.

`ERROR_INVALID_PASSWORD`

The specified password is invalid.

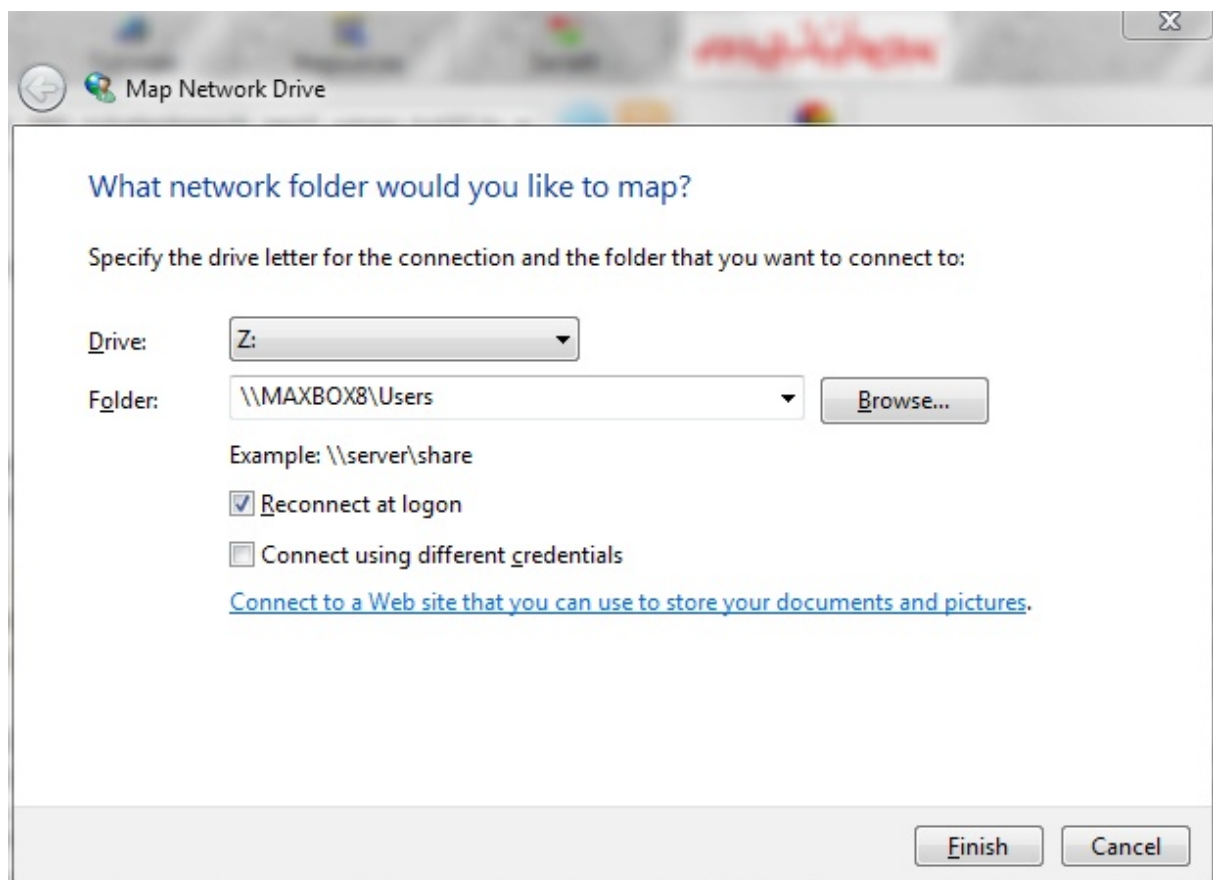
`ERROR_NO_NETWORK`

The network is unavailable.

`ERROR_NOT_ENOUGH_MEMORY`

There is insufficient memory to start the dialog box.

If the user clicks OK in the dialog box, the requested network connection will have been made when the `WNetConnectionDialog` function returns.



What about the password to connect in the dialogue of the API call?

If the function attempts to make a connection and the network provider returns the message `ERROR_INVALID_PASSWORD`, the system prompts the user to enter a password. The system uses the new password in another attempt to make the net connection.

But where comes this function?

They use the windows 32-bit Application programming interface, which basically means interacting with Windows operating systems and libraries such as Windows XP or 7 and the concerning DLL.

So the same call is valuable from a DLL:

```
function MyNETConnect(hw: hwnd; dw: dword): Longint;  
    external 'WNetConnectionDialog@Mpr.dll stdcall';
```

In this tutorial, you will learn how to use an external function with the Win32 API to make wonderful and wise applications. You see the library is called `Mpr.dll` as the external DLL.

And this is how we call the external function:

```
if MyNETConnect(GetForegroundWindow, RESOURCETYPE_DISK)  
    = NO_ERROR then writeln('connect dialog success');
```

You do not always need to know the following datatypes in order to use Win32 API, the lesson here is, that most Win32 types are similar, if not the same as C++ or Object Pascal datatypes. You are free to use standard C++ or Pascal datatypes to express any Win32 types.

A lot of the types in Win32 API, are synonyms and are not really that important to know or to learn cause you can search for type mapping.



It is important to declare your globals and definitions at the top of your program. You can also wrap a API function like this:

```
function TJvNetworkConnect_Execute: Boolean;  
var fconnect: boolean;  
begin  
    //RESOURCETYPE_DISK  
    //WNetDisconnectDialog  
    fconnect:= true;  
    if FConnect then  
        Result:= WnetConnectionDialog(GetForegroundWindow,  
                                         RESOURCETYPE_DISK) = NO_ERROR  
    else  
        Result:= WnetDisconnectDialog(GetForegroundWindow,  
                                         RESOURCETYPE_DISK) = NO_ERROR;  
end;
```

HWNDs are usually used for GUI elements; they represent the actual window or window elements of a program. UINT is an unsigned integer and WPARAM and LPARAM are parameter messages that windows uses to add more data into a message.

In our example it is the first parameter: `GetForegroundWindow`.

But be carefully it is also a function to get the HWND!

`GetForegroundWindow()` retrieves a handle to the foreground window (the window with which the user is currently working). The system assigns a slightly higher priority to the thread that creates the foreground window than it does to other threads.

HWND WINAPI `GetForegroundWindow(void)`;

The return value is a handle to the foreground window. The foreground window can be NULL in certain circumstances, such as when a window is losing activation. So we use in fact 2 API functions:

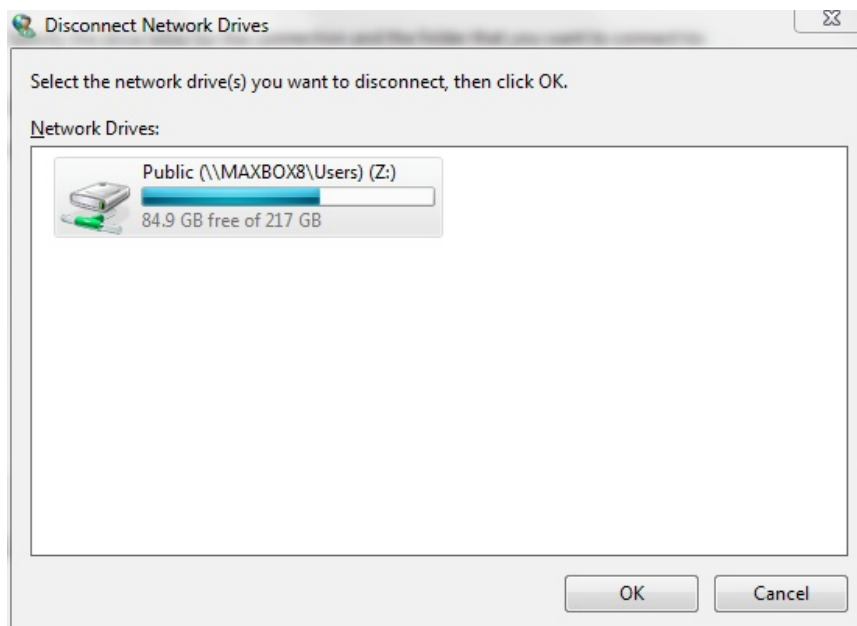
```
WnetConnectionDialog(GetForegroundWindow,  
                     RESOURCETYPE_DISK) = NO_ERROR  
WnetConnectionDialog(GetForegroundWindow,  
                     RESOURCETYPE_DISK) = NO_ERROR
```

Lib: Mpr.dll → `WnetConnectionDialog`

Lib: User32.dll → `GetForegroundWindow`

Then in the end the system uses the new password in another attempt to make a connection.

The `WNetCancelConnection2` function or `WnetDisconnectDialog` cancels an existing network connection. You can also call the function to remove remembered network connections that are not currently connected.



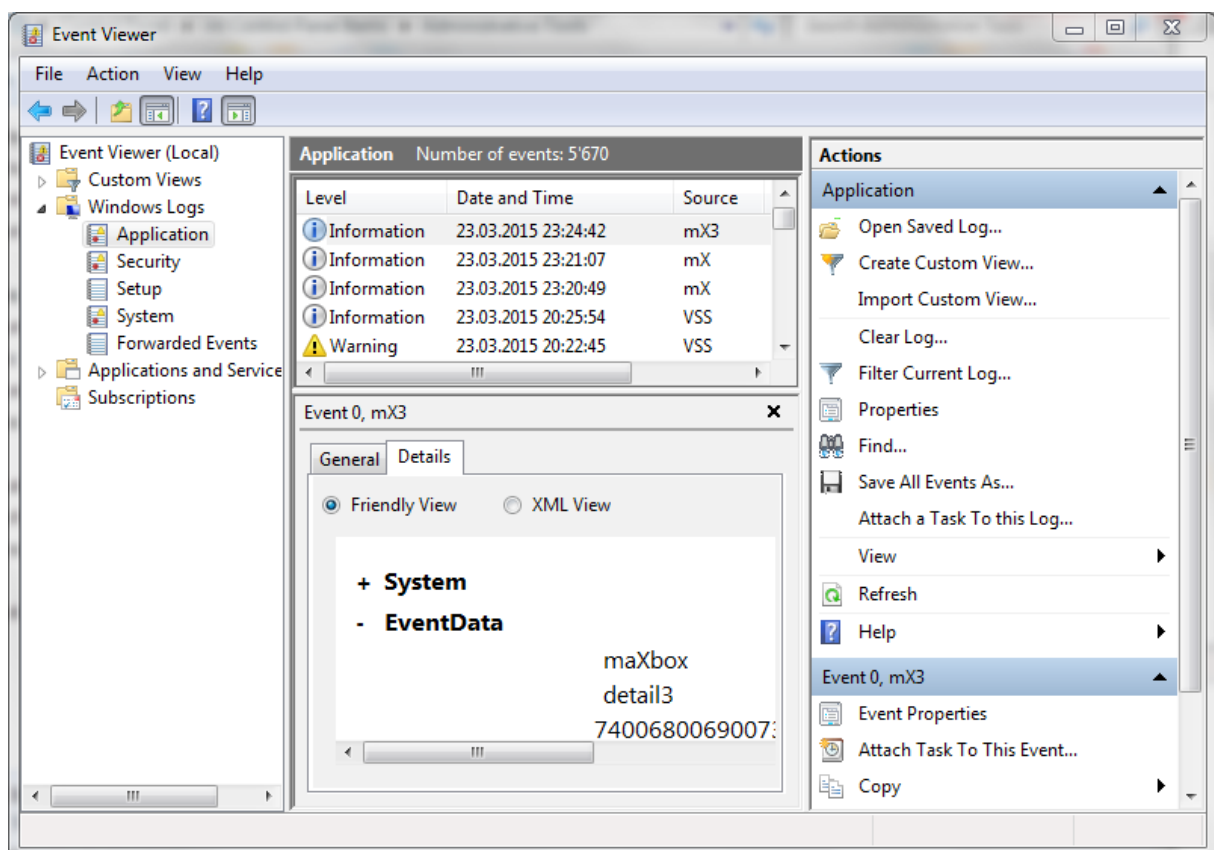
Closes a handle and the object associated with that handle. After being closed, the handle is of course no longer valid. This closes handles associated with access tokens, communications devices, console inputs, console screen buffers, events, files, file mappings, jobs, mail-slots, mutexes, named pipes, processes, semaphores, sockets, and threads.

The Application log (see picture) contains those API or other events logged by applications or programs. For example, a database program might record a file error in the application log. You as a developer decide which events to log.

To log a sound in a multimedia app for e.g. you have to set a song first:

```
playMP3 (mp3path) ;
closeMP3;
DebugOutputString('closeMP3: ' + mp3path);
```

In a shortlink `%windir%\system32\eventvwr.msc /s` you start a log:



👉 Analytic or test events are published in high volume. They describe program operation and indicate problems that can't be handled by user tasks or intervention.

Finally, if a high-resolution performance counter exists on the system, you can use the `QueryPerformanceFrequency` Win API function to express the

frequency, in counts per second. The value of the count is processor dependent!

`QueryPerformanceFrequency` returns the number of "ticks" per seconds - it is the amount that the counter, `QueryPerformanceCounter`, will increment in a second.

In `maXbox` you can also map (connect) or disconnect a network drive without a dialog!:

```
ConnectDrive('Z:', '\\Servername\C', True, True);
```

```
if ConnectDrive('Z:', '\\MAXBOX8\Users\Public', True, True) = NO_ERROR then  
    writeln('Net Share Z:\ Connected');
```


```
DisconnectNetDrive('Z:', True, True, True);
```

Test the script with **F9** / F2 or press Compile. So far now we'll open the API-function example: `580_indystacksearch_geo2_winapi_tut37.txt`

[http://www.softwareschule.ch/examples/587\\_one\\_function\\_assert.txt](http://www.softwareschule.ch/examples/587_one_function_assert.txt)

Hope you did already work with the Starter 28 on DLL Code topics:

[http://www.softwareschule.ch/download/maxbox\\_starter28.pdf](http://www.softwareschule.ch/download/maxbox_starter28.pdf)

 **Conclusion** Stepping through code is one of the most basic debugging operations, yet it still needs to be mentioned here. Several principles are commonly used to govern the process of designing APIs. They proposed the concept of information hiding in 1972. The principle of information hiding is that one may divide software into modules, each of which has a specified interface. The interfaces hide the implementation details of the modules so that users of modules need not understand the complexities inside the modules. These interfaces are APIs, and as a result, APIs should be designed in order to expose only those details of modules that are necessary for clients to know in order to use the modules effectively.

"If the brain were so simple we could understand it, we would be so simple we couldn't." Lyall Watson

Feedback @ [max@kleiner.com](mailto:max@kleiner.com)

Literature: Kleiner et al., Patterns konkret, 2003, Software & Support

[http://www.softwareschule.ch/download/codesign\\_2015.pdf](http://www.softwareschule.ch/download/codesign_2015.pdf)

<http://www.rosseeld.be/DRO/Delphi/Delphi%20WinAPI.htm>

<https://github.com/maxkleiner/maXbox3/releases>

## 1.2 Appendix Event Log Study with Assert

```
PROCEDURE Assert(Cond: boolean; const Msg: string);  
var  
  progSeg,  
  progOfs: word;  
begin  
  asm  
    mov ax, [bp+04]  
    mov es, ax  
    mov ax, word ptr es:0  
    mov progSeg, ax  
    mov ax, [bp+02]  
    mov progOfs, ax  
  end;  
  if (Cond = FALSE) then begin  
    mmDebug.Lines.Add(Msg + ' at location ' +  
      IntToHex(progSeg, 4) + ':' + IntToHex(progOfs, 4) );  
    ShowModal;  
  end;  
end;  
  
// WriteToOSEventLog//  
procedure WriteToOSEventLog(const logName, logCaption, logDetails : UnicodeString;  
  const logRawData : String = "");  
  
var  
  eventSource : THandle;  
  detailsPtr : array [0..1] of PWideChar;  
begin  
  if logName<>" then  
    eventSource:=RegisterEventSourceW(nil, PWideChar(logName))  
  else eventSource:=RegisterEventSourceW(nil,  
  PWideChar(ChangeFileExt(ExtractFileName(ParamStr(0)), ""));  
  if eventSource>0 then begin  
    try  
      detailsPtr[0]:=PWideChar(logCaption);  
      detailsPtr[1]:=PWideChar(logDetails);  
      ReportEventW(eventSource, EVENTLOG_INFORMATION_TYPE, 0, 0, nil,  
        2, Length(logRawData),  
        @detailsPtr, Pointer(logRawData));  
    finally  
      DeregisterEventSource(eventSource);  
    end;  
  end;  
end;  
end;
```

Check the script! Script Example: maxbox3\examples\318\_excel\_export3