Registering for Device Notification

Applications can use the **RegisterDeviceNotification** function to register to receive notification messages from the system. The following example shows how to register for notification of events for the device interfaces which are members of the interface class whose GUID is passed to the function.

C++

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <strsafe.h>
#include <dbt.h>

// This GUID is for all USB serial host PnP drivers, but you can replace it
// with any valid device class guid.
GUID WceusbshGUID = { 0x25dbce51, 0x6c8f, 0x4a72,
                      0x8a,0x6d,0xb5,0x4c,0x2b,0x4f,0xc8,0x35 };

// For informational messages and window titles
PWSTR g_pszAppName;

// Forward declarations
void OutputMessage(HWND hOutWnd, WPARAM wParam, LPARAM lParam);
void ErrorHandler(LPTSTR lpszFunction);

//
// DoRegisterDeviceInterfaceToHwnd
//
BOOL DoRegisterDeviceInterfaceToHwnd(
    IN GUID InterfaceClassGuid,
    IN HWND hWnd,
    OUT HDEVNOTIFY *hDeviceNotify
)
// Routine Description:
//     Registers an HWND for notification of changes in the device interfaces
//     for the specified interface class GUID.

// Parameters:
//     InterfaceClassGuid - The interface class GUID for the device
//         interfaces.

//     hWnd - Window handle to receive notifications.

//     hDeviceNotify - Receives the device notification handle. On failure,
//         this value is NULL.

// Return Value:
//     If the function succeeds, the return value is TRUE.
//     If the function fails, the return value is FALSE.

// Note:
//     RegisterDeviceNotification also allows a service handle be used,
//     so a similar wrapper function to this one supporting that scenario
//     could be made from this template.
{
    DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;

    ZeroMemory( &NotificationFilter, sizeof(NotificationFilter) );
    NotificationFilter.dbcc_size = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
    NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
    NotificationFilter.dbcc_classguid = InterfaceClassGuid;

    *hDeviceNotify = RegisterDeviceNotification(
        hWnd,                       // events recipient
        &NotificationFilter,        // type of device
        DEVICE_NOTIFY_WINDOW_HANDLE // type of recipient handle
        );

    if ( NULL == *hDeviceNotify )
    {
        ErrorHandler(TEXT("RegisterDeviceNotification"));
        return FALSE;
    }

    return TRUE;
}

//
// MessagePump
//
void MessagePump(
```

```
    HWND hWnd
)
// Routine Description:
//     Simple main thread message pump.
//

// Parameters:
//     hWnd - handle to the window whose messages are being dispatched

// Return Value:
//     None.
{
    MSG msg;
    int retVal;

    // Get all messages for any window that belongs to this thread,
    // without any filtering. Potential optimization could be
    // obtained via use of filter values if desired.

    while( (retVal = GetMessage(&msg, NULL, 0, 0)) != 0 )
    {
        if ( retVal == -1 )
        {
            ErrorHandler(TEXT("GetMessage"));
            break;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}

//
// WinProcCallback
//
INT_PTR WINAPI WinProcCallback(
                              HWND hWnd,
                              UINT message,
                              WPARAM wParam,
                              LPARAM lParam
                              )
// Routine Description:
//     Simple Windows callback for handling messages.
//     This is where all the work is done because the example
//     is using a window to process messages. This logic would be handled
//     differently if registering a service instead of a window.

// Parameters:
//     hWnd - the window handle being registered for events.

//     message - the message being interpreted.

//     wParam and lParam - extended information provided to this
//          callback by the message sender.

//     For more information regarding these parameters and return value,
//     see the documentation for WNDCLASSEX and CreateWindowEx.
{
    LRESULT lRet = 1;
    static HDEVNOTIFY hDeviceNotify;
    static HWND hEditWnd;
    static ULONGLONG msgCount = 0;

    switch (message)
    {
    case WM_CREATE:
        //
        // This is the actual registration., In this example, registration
        // should happen only once, at application startup when the window
        // is created.
        //
        // If you were using a service, you would put this in your main code
        // path as part of your service initialization.
        //
        if ( ! DoRegisterDeviceInterfaceToHwnd(
                        WceusbshGUID,
                        hWnd,
                        &hDeviceNotify) )
        {
            // Terminate on failure.
            ErrorHandler(TEXT("DoRegisterDeviceInterfaceToHwnd"));
            ExitProcess(1);
        }
```

```
        //
        // Make the child window for output.
        //
        hEditWnd = CreateWindow(TEXT("EDIT"),// predefined class
                                NULL,        // no window title
                                WS_CHILD | WS_VISIBLE | WS_VSCROLL |
                                ES_LEFT | ES_MULTILINE | ES_AUTOVSCROLL,
                                0, 0, 0, 0,  // set size in WM_SIZE message
                                hWnd,        // parent window
                                (HMENU)1,    // edit control ID
                                (HINSTANCE) GetWindowLong(hWnd, GWL_HINSTANCE),
                                NULL);       // pointer not needed

        if ( hEditWnd == NULL )
        {
            // Terminate on failure.
            ErrorHandler(TEXT("CreateWindow: Edit Control"));
            ExitProcess(1);
        }
        // Add text to the window.
        SendMessage(hEditWnd, WM_SETTEXT, 0,
            (LPARAM)TEXT("Registered for USB device notification...\n"));

        break;

    case WM_SETFOCUS:
        SetFocus(hEditWnd);

        break;

    case WM_SIZE:
        // Make the edit control the size of the window's client area.
        MoveWindow(hEditWnd,
                   0, 0,                  // starting x- and y-coordinates
                   LOWORD(lParam),        // width of client area
                   HIWORD(lParam),        // height of client area
                   TRUE);                 // repaint window

        break;

    case WM_DEVICECHANGE:
    {
        //
        // This is the actual message from the interface via Windows messaging.
        // This code includes some additional decoding for this particular device type
        // and some common validation checks.
        //
        // Note that not all devices utilize these optional parameters in the same
        // way. Refer to the extended information for your particular device type
        // specified by your GUID.
        //
        PDEV_BROADCAST_DEVICEINTERFACE b = (PDEV_BROADCAST_DEVICEINTERFACE) lParam;
        TCHAR strBuff[256];

        // Output some messages to the window.
        switch (wParam)
        {
        case DBT_DEVICEARRIVAL:
            msgCount++;
            StringCchPrintf(
                strBuff, 256,
                TEXT("Message %d: DBT_DEVICEARRIVAL\n"), msgCount);
            break;
        case DBT_DEVICEREMOVECOMPLETE:
            msgCount++;
            StringCchPrintf(
                strBuff, 256,
                TEXT("Message %d: DBT_DEVICEREMOVECOMPLETE\n"), msgCount);
            break;
        case DBT_DEVNODES_CHANGED:
            msgCount++;
            StringCchPrintf(
                strBuff, 256,
                TEXT("Message %d: DBT_DEVNODES_CHANGED\n"), msgCount);
            break;
        default:
            msgCount++;
            StringCchPrintf(
                strBuff, 256,
                TEXT("Message %d: WM_DEVICECHANGE message received, value %d unhandled.\n"),
                msgCount, wParam);
            break;
        }
```

```
            OutputMessage(hEditWnd, wParam, (LPARAM)strBuff);
        }
                break;
        case WM_CLOSE:
            if ( ! UnregisterDeviceNotification(hDeviceNotify) )
            {
               ErrorHandler(TEXT("UnregisterDeviceNotification"));
            }
            DestroyWindow(hWnd);
            break;

        case WM_DESTROY:
            PostQuitMessage(0);
            break;

        default:
            // Send all other messages on to the default windows handler.
            lRet = DefWindowProc(hWnd, message, wParam, lParam);
            break;
        }

        return lRet;
    }

    #define WND_CLASS_NAME TEXT("SampleAppWindowClass")

    //
    // InitWindowClass
    //
    BOOL InitWindowClass()
    // Routine Description:
    //      Simple wrapper to initialize and register a window class.

    // Parameters:
    //     None

    // Return Value:
    //     TRUE on success, FALSE on failure.

    // Note:
    //     wndClass.lpfnWndProc and wndClass.lpszClassName are the
    //     important unique values used with CreateWindowEx and the
    //     Windows message pump.
    {
        WNDCLASSEX wndClass;

        wndClass.cbSize = sizeof(WNDCLASSEX);
        wndClass.style = CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
        wndClass.hInstance = reinterpret_cast<HINSTANCE>(GetModuleHandle(0));
        wndClass.lpfnWndProc = reinterpret_cast<WNDPROC>(WinProcCallback);
        wndClass.cbClsExtra = 0;
        wndClass.cbWndExtra = 0;
        wndClass.hIcon = LoadIcon(0,IDI_APPLICATION);
        wndClass.hbrBackground = CreateSolidBrush(RGB(192,192,192));
        wndClass.hCursor = LoadCursor(0, IDC_ARROW);
        wndClass.lpszClassName = WND_CLASS_NAME;
        wndClass.lpszMenuName = NULL;
        wndClass.hIconSm = wndClass.hIcon;


        if ( ! RegisterClassEx(&wndClass) )
        {
            ErrorHandler(TEXT("RegisterClassEx"));
            return FALSE;
        }
        return TRUE;
    }

    //
    // main
    //

    int __stdcall _tWinMain(
                        HINSTANCE hInstanceExe,
                        HINSTANCE, // should not reference this parameter
                        PTSTR lpstrCmdLine,
                        int nCmdShow
                        )
    {
    //
    // To enable a console project to compile this code, set
    // Project->Properties->Linker->System->Subsystem: Windows.
    //

        int nArgC = 0;
```

```
    PWSTR* ppArgV = CommandLineToArgvW(lpstrCmdLine, &nArgC);
    g_pszAppName = ppArgV[0];

    if ( ! InitWindowClass() )
    {
        // InitWindowClass displays any errors
        return -1;
    }

    // Main app window

    HWND hWnd = CreateWindowEx(
                    WS_EX_CLIENTEDGE | WS_EX_APPWINDOW,
                    WND_CLASS_NAME,
                    g_pszAppName,
                    WS_OVERLAPPEDWINDOW, // style
                    CW_USEDEFAULT, 0,
                    640, 480,
                    NULL, NULL,
                    hInstanceExe,
                    NULL);

    if ( hWnd == NULL )
    {
        ErrorHandler(TEXT("CreateWindowEx: main appwindow hWnd"));
        return -1;
    }

    // Actually draw the window.

    ShowWindow(hWnd, SW_SHOWNORMAL);
    UpdateWindow(hWnd);

    // The message pump loops until the window is destroyed.

    MessagePump(hWnd);

    return 1;
}

//
// OutputMessage
//
void OutputMessage(
    HWND hOutWnd,
    WPARAM wParam,
    LPARAM lParam
)
// Routine Description:
//     Support routine.
//     Send text to the output window, scrolling if necessary.

// Parameters:
//     hOutWnd - Handle to the output window.
//     wParam  - Standard windows message code, not used.
//     lParam  - String message to send to the window.

// Return Value:
//     None

// Note:
//     This routine assumes the output window is an edit control
//     with vertical scrolling enabled.

//     This routine has no error checking.
{
    LRESULT   lResult;
    LONG      bufferLen;
    LONG      numLines;
    LONG      firstVis;

    // Make writable and turn off redraw.
    lResult = SendMessage(hOutWnd, EM_SETREADONLY, FALSE, 0L);
    lResult = SendMessage(hOutWnd, WM_SETREDRAW, FALSE, 0L);

    // Obtain current text length in the window.
    bufferLen = SendMessage (hOutWnd, WM_GETTEXTLENGTH, 0, 0L);
    numLines = SendMessage (hOutWnd, EM_GETLINECOUNT, 0, 0L);
    firstVis = SendMessage (hOutWnd, EM_GETFIRSTVISIBLELINE, 0, 0L);
    lResult = SendMessage (hOutWnd, EM_SETSEL, bufferLen, bufferLen);

    // Write the new text.
    lResult = SendMessage (hOutWnd, EM_REPLACESEL, 0, lParam);

    // See whether scrolling is necessary.
```

```
    if (numLines > (firstVis + 1))
    {
        int        lineLen = 0;
        int        lineCount = 0;
        int        charPos;

        // Find the last nonblank line.
        numLines--;
        while(!lineLen)
        {
            charPos = SendMessage(
                hOutWnd, EM_LINEINDEX, (WPARAM)numLines, 0L);
            lineLen = SendMessage(
                hOutWnd, EM_LINELENGTH, charPos, 0L);
            if(!lineLen)
                numLines--;
        }
        // Prevent negative value finding min.
        lineCount = numLines - firstVis;
        lineCount = (lineCount >= 0) ? lineCount : 0;

        // Scroll the window.
        lResult = SendMessage(
            hOutWnd, EM_LINESCROLL, 0, (LPARAM)lineCount);
    }

    // Done, make read-only and allow redraw.
    lResult = SendMessage(hOutWnd, WM_SETREDRAW, TRUE, 0L);
    lResult = SendMessage(hOutWnd, EM_SETREADONLY, TRUE, 0L);
}

//
// ErrorHandler
//
void ErrorHandler(
   LPTSTR lpszFunction
)
// Routine Description:
//     Support routine.
//     Retrieve the system error message for the last-error code
//     and pop a modal alert box with usable info.

// Parameters:
//     lpszFunction - String containing the function name where
//     the error occurred plus any other relevant data you'd
//     like to appear in the output.

// Return Value:
//     None

// Note:
//     This routine is independent of the other windowing routines
//     in this application and can be used in a regular console
//     application without modification.
{

    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM |
        FORMAT_MESSAGE_IGNORE_INSERTS,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    // Display the error message and exit the process.

    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR)lpMsgBuf)
                  + lstrlen((LPCTSTR)lpszFunction)+40)
                  * sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(TCHAR),
        TEXT("%s failed with error %d: %s"),
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, g_pszAppName, MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

This example uses a window handle for device change notifications and contains a large amount of code to support the windowing mechanisms. An application intended to be installed as a Windows service would contain different support code.

The application will receive the **WM_DEVICECHANGE** message whenever a USB device interface event notification is sent.

The system broadcasts a set of default device change events to all applications and services. You do not need to register to receive these default events. For details, see the Remarks section in **RegisterDeviceNotification**.

The following is an example makefile for building the previous example code using nmake.exe from the SDK or Visual Studio command prompt.

```
proj = main

all: $(proj).exe

# Update the object file if necessary

$(proj).obj: $(proj).cpp
  cl -c  /D_UNICODE /DUNICODE -DWIN32 -D_WIN32 -DNDEBUG -GS -D_X86_=1 -D_WIN32_WINNT=0x0501 -DCRTAPI1=_cdecl -DCRTAPI2=_cdecl -D_MT -D_DLL -MD $*

# Update the executable file if necessary

$(proj).exe: $(proj).obj
  link -release -incremental:no -nologo -subsystem:windows,5.01 $(proj).obj kernel32.lib user32.lib advapi32.lib gdi32.lib shell32.lib -out:$(pro
```

Related topics

Device Notifications

Community Additions

DBT_CUSTOMEVENT instead of DBT_DEVICEQUERYREMOVE

On Windows7 x64 with all updates today I experienced the following:

When I connected my Samsung mobile phone, 2 drives appeared: internal memory and external one, say X: and Y:. My application opens "\\.\Y:" and calls RegisterDeviceNotification passing DEVICE_NOTIFY_WINDOW_HANDLE and DEV_BROADCAST_HANDLE.

When I click "Safely remove USB device", the system DOESN'T SEND DBT_DEVICEQUERYREMOVE, and sends DBT_CUSTOMEVENT with eventguid GUID_IO_VOLUME_LOCK instead.

Probably this is caused by the fact that much more than my "Y:" will be ejected. Anyway the actions expected from application by the system are the same: close all handles.

Mike Makarov
9/3/2014

Cannot capture disconnect/reconnect of keyboard.

When testing this code and disconnecting/reconnecting a keyboard I only get the DB_DEVNODES_CHANGED event, which does not provide any more detail about the specific device. Several websites stated that `4D36E96B-E325-11CE-BFC1-08002BE10318` is the class guid for keyboards, so I tried to register device notification for this guid -- still nothing more than `DB_DEVNODES_CHANGED`. When I use `IDirectInput8::EnumDevices(DI8DEVCLASS_KEYBOARD, ...)` I can get the specific guid for the keyboard, but using the guidInstance or guidProduct (which is the same) has no effect.

I have tried to simply keep track of the connection state of my keyboard using `IDirectInput8::EnumDevices`, however, I continue to get callbacks for the keyboard after I disconnect it and there is no change to any of the state variables.

I have looked through all the other API's and it seems there is nothing for determining when a keyboard is connnected or disconnected.

Am I missing something obvious? How can I determine when my keyboard is connected/disconnected?


JonathanAH
3/29/2012

Interop

This is an unmanaged API, as has already been pointed out; nonetheless, if Microsoft is serious about C# and the .NET framework, they should be providing managed code examples along with any interop required. Interop is a pain in the neck and easy to get wrong; needed frequently enough to be a necessary part of most any project I've needed to code, yet not so often that one retains a proficiency at it.

Microsoft has about 1,000 developers working on VS2010 alone. You'd think they could afford a few developers to create managed wrappers and examples.


Dave Ruske
2/18/2011

Registering for Device Notification (Windows) using C#

Here are a couple good articles on how to do this in C#:

http://www.codeproject.com/KB/system/DriveDetector.aspx

http://www.developerfusion.com/article/84338/making-usb-c-friendly/


pcpro1789
11/2/2010

Why no C#

You won't see a C# example here because this is a Windows API, it's not part of the .NET framework. To call this from C# you need to use pinvoke, and I think providing pinvoke wrappers for all the various Windows APIs falls outside the scope of normal MSDN documentation.


MichaC
10/15/2010

C#

Where is the C# code? I filtered on C# and this is code for c/C++

Somebody?

Thanks


beast888
9/6/2010