Written by Tom Bell

Detecting USB Device Insertion and Removal Using Windows API

When I needed to know how to detect USB device insertion and removal, I was developing an application for backing up USB devices. I had to research the methods using the Microsoft Developer Network. The method described in this guide can be used whether you're developing a Windowed application, or a Windows service. I will outline how to detect the devices in a Windowed application and Windows service.

First of all we are going to look at how we would go about creating a user-mode application to detect a hardware device change on the system, for example plugging in a USB disk drive (this refers to USB hard disk drives, and USB thumb and flash drives), plugging in an iPod, or any other USB device. You can also do things such as disabling the device, but this is beyond the scope of this guide.

How Do I Detect a Hardware Device Change?

It is very easy to detect this change, because the operating system will send a *WM_DEVICECHANGE* message to the application when a device change is detected. All we need to do is handle this message in the window procedure of the application. When the window procedure is called the parameters passed will be as follows.

Parameter	Description
HWND hWnd	Handle to the window
UINT uiMessage	WM_DEVICECHANGE
WPARAM wParam	Device-change event
LPARAM lParam	Event-specific data

This means, *hWnd* will be the handle to our window, *uiMessage* will be the window message *WM_DEVICECHANGE*, *wParam* will be the device-change event such as *DBT_DEVICEARRIVAL* or *DBT_DEVICEREMOVECOMPLETE*, and finally *lParam* is a pointer to the device broadcast header. Below is an example of how you could implement a handler for *WM_DEVICECHANGE* in code section 1.0.

Code Section 1.0

```
LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg)
  {
    case WM_DEVICECHANGE:
    {
      PDEV_BROADCAST_HDR pHdr = (PDEV_BROADCAST_HDR) lParam;

      switch (wParam)
      {
        case DBT_DEVICEARRIVAL:
          MessageBox(hWnd, "A device has been inserted.", "USB Notice", MB_OK);
        break;

        case DBT_DEVICEREMOVECOMPLETE:
          MessageBox(hWnd, "A device has been removed.", "USB Notice", MB_OK);
        break;
      }
    }
    break;

    default:
```

```
      return DefWindowProc(hWnd, uiMsg, wParam, lParam);
    break;
  }

  return 0;
}
```

This code will display a message box when a new device is inserted or removed from the system, with an appropriate message. You can take it further by detecting the device type which is stored in the *dbch_devicetype* member of the *PDEV_BROADCAST_HEADER*.

There are a couple of problems which you may encounter. The first is that the operating system will only send the *WM_DEVICECHANGE* to applications with a top-level window. The second is that the operating system will send the *WM_DEVICECHANGE* message on port and volume changes only. This is not that much of a problem since you will know when you mount and unmount an extra disk drive.

Fortunately for us, the Windows API provides a way for us to receive notification of other types of device changes. You can use this method if you are running as a Windows service, or don't have a top-level window. We can use the function *RegisterDeviceNotification(...)* to be notified upon interface change. An example of registering a device notification is shown below in code section 2.0.

Code Section 2.0

```
DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;
HDEVNOTIFY hDeviceNotify = NULL;

static const GUID GuidDevInterfaceDisk =
{
  0x53f56307, 0xb6bf, 0x11d0, { 0x94, 0xf2, 0x00, 0xa0, 0xc9, 0x1e, 0xfb, 0x8b }
};

ZeroMemory(&NotificationFilter, sizeof(NotificationFilter));

NotificationFilter.dbcc_size = sizeof(DEV_BROADCADT_DEVICEINTERFACE);
NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
NotificationFilter.dbcc_classguid = GuidDevInterfaceDisk;

hDeviceNotify = RegisterDeviceNotification(hWnd, &NotificationFilter
DEVICE_NOTIFY_WINDOW_HANDLE);

if (hDeviceNotify == NULL)
{
  // Handle the error...
}
```

Plug and Play (PnP) devices are typically associated with two different GUIDs, a device interface GUID, and a device class GUID. A device class GUID defines a broad category of devices. When you look in the Windows Device Manager, it is ordered by the type of devices. Each of those devices is a device class and each of those classes is identified by a device class GUID.

A device interface GUID specifies a particular input/output interface contract. Every instance of the device interface GUID is expected to support the same basic set of inputs/outputs. The device interface GUID is what the device driver will register and enable or disabled based on the PnP state.

In the above code we just use the GUID for a disk device. Some of the common GUIDs for device interface classes are listed below.

Device Interface Name	Device Interface Class GUID
USB Raw Device	{a5dcbf10-6530-11d2-901f-00c04fb951ed}
Disk Device	{53f56307-b6bf-11d0-94f2-00a0c91efb8b}
Human Interface Device (HID)	{4d1e55b2-f16f-11cf-88cb-001111000030}
Network Card	{784126bf-4190-11d4-b5c2-00c04f687a67}

You can use a simple *for* loop to register multiple device notifications for each GUID. An example is shown below in code section 3.0.

Code Section 3.0

```
DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;
HDEVNOTIFY hDeviceNotify = NULL;

static const GUID GuidDevInterfaceList[] =
{
  { 0xa5dcbf10, 0x6530, 0x11d2, { 0x90, 0x1f, 0x00, 0xc0, 0x4f, 0xb9, 0x51, 0xed } },
  { 0x53f56307, 0xb6bf, 0x11d0, { 0x94, 0xf2, 0x00, 0xa0, 0xc9, 0x1e, 0xfb, 0x8b } },
  { 0x4d1e55b2, 0xf16f, 0x11Cf, { 0x88, 0xcb, 0x00, 0x11, 0x11, 0x00, 0x00, 0x30 } },
  { 0xad498944, 0x762f, 0x11d0, { 0x8d, 0xcb, 0x00, 0xc0, 0x4f, 0xc3, 0x35, 0x8c } }
};

ZeroMemory(&NotificationFilter, sizeof(NotificationFilter));

NotificationFilter.dbcc_size = sizeof(DEV_BROADCADT_DEVICEINTERFACE);
NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;

for (int i = 0; i < sizeof(GuidDevInterfaceList); i++)
{
  NotificationFilter.dbcc_classguid = GuidDevInterfaceList[i];

  hDeviceNotify = RegisterDeviceNotification(hWnd, &NotificationFilter
DEVICE_NOTIFY_WINDOW_HANDLE);
  if (hDeviceNotify == NULL)
  {
    // Handle the error...
  }
}
```

This will register a device notification for each of the device interface class GUIDs. Now our application will receive the *WM_DEVICECHANGE* window message even if it is not a top-level window.

If you are doing this as a Windows service, there are some significant changes you will need to make to the above code. When you register a device notification you will need to use the constant *DEVICE_NOTIFY_SERVICE_HANDLE* instead of *DEVICE_NOTIFY_WINDOW_HANDLE*. Example code for registering a device notification for a Windows service is shown below in code section 4.0.

Code Section 4.0

```
SERVICE_STATUS_HANDLE hServiceStatus = NULL;

hServiceStatus = RegisterServiceCtrlHandlerEx("ServiceName", (LPHANDLER_FUNCTION_EX)
HandlerProcedure, 0);

if (hServiceStatus == NULL)
{
  // Handle the error...
}

DEV_BROADCAST_DEVICEINTERFACE NotificationFilter;
HDEVNOTIFY hDeviceNotify = NULL;

static const GUID GuidDevInterfaceDisk =
{
```

```
  0x53f56307, 0xb6bf, 0x11d0, { 0x94, 0xf2, 0x00, 0xa0, 0xc9, 0x1e, 0xfb, 0x8b }
};

ZeroMemory(&NotificationFilter, sizeof(NotificationFilter));

NotificationFilter.dbcc_size = sizeof(DEV_BROADCADT_DEVICEINTERFACE);
NotificationFilter.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
NotificationFilter.dbcc_classguid = GuidDevInterfaceDisk;

hDeviceNotify = RegisterDeviceNotification(hServiceStatus, &NotificationFilter
DEVICE_NOTIFY_SERVICE_HANDLE);

if (hDeviceNotify == NULL)
{
  // Handle the error...
}
```

Now the message *SERVICE_CONTROL_DEVICEEVENT* will be sent the handler procedure passed as a parameter to *RegisterServiceCtrlHandlerEx(...)*. We need to handle the service control message *SERVICE_CONTROL_DEVICEEVENT*, which is similar to *WM_DEVICECHANGEI*. When the service control handler is called the parameters that will be passed are as follows.

Parameter	Description
DWORD dwOpcode	SERVICE_CONTROL_DEVICEEVENT
DWORD evtype	Device-change event
DWORD evdata	Event-specific data
PVOID Context	User-defined data

With these parameters in mind, we can handle the device change event. Handling the event is similar to the method used in a windowed application. Except the parameters have different names. Below is an example of how to handle the device change event in code section 5.0.

Code Section 5.0

```
void WINAPI HandlerProcedure(DWORD dwOpcode, DWORD evtype, PVOID evdata, PVOID
Context)
{
  switch (dwOpcode)
  {
    case SERVICE_CONTROL_DEVICEEVENT:
      switch (evtype)
      {
        case DBT_DEVICEARRIVAL:
          // Handle device arrival here...
        break;

        case DBT_DEVICEREMOVECOMPLETE:
          // Handle device removal here...
        break;
      }
    break;
  }
}
```

You can handle the different device-events accordingly. Now you should understand the basics of detecting whether a USB device has been inserted or removed. As an extra last minute tip, if the device plugged in is a volume, you can get the drive letter of the volume which has been inserted or removed. You can find this in the *dbcv_unitmask* of a *PDEV_BROADCAST_VOLUME* structure. Example code for this is shown below in code section 6.0.

Code Section 6.0

```
char GetDriveLetter(unsigned long ulUnitMask)
{
  for (char c = 0; c < 26; c++)
  {
    if (ulUnitMask & 0x01)
    {
      break;
    }

    ulUnitMask = ulUnitMask >> 1;
  }

  return (c + 'A');
}
```

This function is passed the member *dbcv_unitmask* of a *PDEV_BROADCAST_VOLUME* structure. The function returns the drive letter as a *char*. Below is an example of using this function in the window procedure function of a windowed application to get the drive letter, in code section 7.0.

Code Section 7.0

```
LRESULT WINAPI WindowProcedure(HWND hWnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
  switch (uiMsg)
  {
    case WM_DEVICECHANGE:
    {
      PDEV_BROADCAST_HDR pHdr = (PDEV_BROADCAST_HDR) lParam;

      switch (wParam)
      {
        case DBT_DEVICEARRIVAL:
          if (pHdr->dhch_devicetype == DBT_DEVTYP_VOLUME)
          {
            PDEV_BROADCAST_VOLUME pVol = (PDEV_BROADCAST_VOLUME) pHdr;
            char szMessage[80];

            char cDriveLetter = GetDriveLetter(pVol->dbcv_unitmask);

            wsprintf(szMessage, "Device '%c:' has been inserted.", cDriveLetter);

            MessageBox(hWnd, szMessage, "USB Notice", MB_OK);
          }
        break;
      }
    }
    break;
  }

  return 0;
}
```

This would display a message box which displays a message saying "Device '%c:' has been inserted" where *%c* is the drive letter of the device.

Unfortunately, this will trigger on any disk device being inserted. For example CD/DVD disks, USB disks, etc. I did come up with a small hack to get around this. I found out that *WM_DEVICECHANGE* is sent to the window procedure twice when you plug a USB disk drive in. The first time *WM_DEVICECHANGE* is sent, the *dhch_devicetype* is *DBT_DEVTYP_DEVICEINTERFACE*, we can check for this device type first, and set a flag variable to true. The second time *WM_DEVICECHANGE* is received, *dhch_devicetype* is *DBT_DEVTYP_VOLUME*. So we can check for this device type and whether the flag for a

USB device is set to true. After you handle the new volume, you set the flag back to false.

For more information about the *WM_DEVICECHANGE* window message visit:
http://msdn2.microsoft.com/en-us/library/aa363480.aspx

For more Information about the *SERVICE_CONTROL_DEVICEEVENT* message visit:
http://msdn2.microsoft.com/en-us/library/ms683241.aspx