Technical Specification

Application design for correctly handling Plug and Play in Windows Systems

Option Confidential

About this document

Overview and Purpose

This document is aimed at application writers wishing to open devices that are subject to the Windows Plug and Play system.

Confidentiality

All data and information contained or disclosed by this document is confidential and proprietary of Option nv, and all rights therein are expressly reserved. By accepting this document, the recipient agrees that this information is held in confidence and in trust and will not be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without prior and written permission of Option nv.

Version History

Date	Version	Author(s)	Revision(s)	Remarks
Oct 22, 2007	V001ext	M. Sykes		Initial version
Dec 12, 2007	V002ext	M. Sykes		Elaborations on DBT_DEVICEREMOVECOMPLETE.

Table of contents

1 INTRODUCTION

Removing devices from a running system is tough on the system. It has a big impact on the device drivers, but also on applications that are accessing that device when it is removed.

This document is an aid to application writers to handle these events correctly.

2 REFERENCES

Ref	Document	

3 MESSAGES GENERATED BY THE SYSTEM

The system generates WM_DEVICE change messages to notify applications that a device state has changed.

4 REGISTERING FOR THOSE MESSAGES

To get these messages an application has to register for them using RegisterDeviceNotification().

The application then needs to map the message into a handler. This is language dependent.

Using RegisterDeviceNotification() is a two stage process. First the application registers for events by Interface, then by handle.

So, to register for events on our Network device, we specify the Network device GUID.

```
DEFINE_GUID(GUID_NDIS_LAN_CLASS,  0xad498944, 0x762f, 0x11d0, 0x8d,
0xcb, 0x00, 0xc0, 0x4f, 0xc3, 0x35, 0x8c);
```

```
ZeroMemory( &devNotification,           sizeof(devNotification) );
devNotification.dbcc_size       = sizeof(DEV_BROADCAST_DEVICEINTERFACE);
devNotification.dbcc_devicetype = DBT_DEVTYP_DEVICEINTERFACE;
devNotification.dbcc_classguid         = GUID_NDIS_LAN_CLASS;

hInterfaceNotification = RegisterDeviceNotification(this->GetSafeHwnd(),
                                                    &devNotification,
                                                    DEVICE_NOTIFY_WINDOW_HANDLE);
```

5 DEVICE ARRIVAL MESSAGE

The application then gets a WM_DEVICECHANGE message with a message type of DBT_DEVICEARRIVAL when the device is inserted.

With the message is a PDEV_BROADCAST_DEVICEINTERFACE structure.

The application should then check the name of the device (dbcc_name) to make sure it is one we are interested in.

GetDeviceDescription(p->dbcc_name, DeviceName)); DeviceName is a CString.

Accompanying this document is a zip file containing source code for this function.

If this is our device we need to register for device events a second time, but this time by handle:

```
m_hDevice = CreateFile(p->dbcc_name,
                       MAXIMUM_ALLOWED  ,
                       0,
                       NULL,
                       OPEN_EXISTING,
                       0,
                       NULL);

if(m_hDevice == INVALID_HANDLE_VALUE)
{
        doerror();
        break;
}

ZeroMemory(&filter,    sizeof(filter));
filter.dbch_size       = sizeof(filter);
filter.dbch_devicetype = DBT_DEVTYP_HANDLE;
filter.dbch_handle     = m_hDevice;

hHandleNotification    = RegisterDeviceNotification(GetSafeHwnd(),
                       &filter,
                       DEVICE_NOTIFY_WINDOW_HANDLE);
```

Doing this second registration allows the application to receive DBT_DEVICEQUERYREMOVE messages.

Not too that the handle returned by the CreateFile function can be used for general IO calls, such as WriteFile() and DeviceOPControl();

the application can also open any symbolic link name on the device, such as "GTNDIS0" and perform IO on it.

6 DEVICE QUERY REMOVAL MESSAGE

The application receives a DBT_DEVICEQUERYREMOVE when the user does a safe remove.

If the application wants to allow this it must deregister for device events by handle by calling

UnregisterDeviceNotification(hHandleNotification);

It must also call CloseHandle(m_hDevice); on the handle it got calling CreateFile() in the DBT_DEVICEARIVAL handler.

It must also close any handles it opened on the symbolic link.

7 DEVICE REMOVAL

The application gets a DBT_DEVICEREMOVECOMPLETE when the card is finally removed, and when the card is surprise removed.

The application must deregister for notification by handle and close any handles on the device the same way it does for DBT_QUERYREMOVE.

*NOTE: Depending on the class of device you have registered for events on you might get either a DBT_DEVICEREMOVECOMPLETE by INTERFACE, or by HANDLE.

With the Network device class GUID, you will get it by INTERFACE, with other classes (our own bespoke bus class GUID) we get it by HANDLE.

So it is best to handle both types of message in the DBT_DEVICEREMOVECOMPLETE handler and close all open handles on the device.

8 SUMARY

That is all there is to it, and if an application follows this it will always know when the device is there or not, and when it can and cant access the device.

Source code is available that demonstrates this, it is in GtmNicApp.zip.