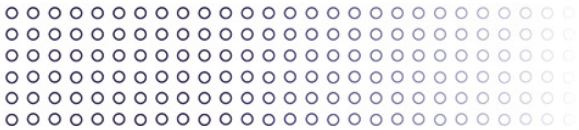

OPTION
WIRELESS TECHNOLOGY

Technical Specification

Application design for correctly handling Plug and Play in Windows Systems

Option Confidential

Option nv, Gaston Geenslaan 14, 3001 Leuven - Belgium - Tel +32 16 317 411 - Fax +32 16 207 164

About this document

Overview and Purpose

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

Confidentiality

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.
Apr 29 2008	v003ext	M. Sykes		small improvements regarding device types on our devices

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_DEVICECHANGE 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() which is a two stage process, first by device interface GUID and then by handle.

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

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);

Finding out the GUIDS for device classes is almost impossible. They are almost totally undocumented even though Microsoft want applications to use this method of handling PnP devices.

You can often intuit the correct GUID though by looking in the registry at HKLM\Sys\CCS\Control\DeviceClasses.

```
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_H
                                                    ANDLE);
```

When a network device is inserted into the system a DBT_DEVICEARIVAL message is generated and posted to all registered apps and services.

5 DEVICE ARRIVAL MESSAGE

With the message is a PDEV_BROADCAST_DEVICEINTERFACE structure. This contains a system generated symbolic link name for the device, p->dbcc_name, which takes the form of "##?#OPTIONBUS#GTS_FF_NET#6&33055f51&0&#{ad498944-762f-11d0-8dcb-00c04fc3358c}{ 8B90C8C7-1244-4788-A590-30CDB0EC9B4C}"

As horrendous as this looks, it is usefull. For, although this friendly name relates to an Ndis device, and so isn't usefull to an application, if we had registered for events with a COM device interface GUID you could directly pass this system created symbolic link to CreateFile(); and use the returned handle to do all the same kinds of IO that you would do if you had opened "\\\\.\\COM11" for example.

This is very useful then. Your app no longer needs to know the COM number.

The application should then check the name of the device associated with the symbolic link (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.

Now, at this stage you have the actual Ndis device open but an app cant do IO directly with an Ndis device, Ndis disallows it. So the Ndis driver creates another device that can be opened by an app. It names this device "GtNdis*x*" Where 'x' is a numeric number from zero onwards incrementing automatically each time a new card is inserted in the machine. Generally of course, this will be "GtNdis0".

You need to call CreateFile() then using GtNdis0 etc for the first parameter.

If you call the IOCTL_GT_NDIS_GPRS_GET_NET_CFG_ID on the gtndis'x' device you will be given a GUID. This is actually the same as the second GUID in the Ndis device handle passed to you in the DBT_DEVICEARRIVAL message and is the only way to associate a particular GtNdis'x' device with a particular Ndis device.

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 GtNdis'x' 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 SUMMARY

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.