

**Jungo DriverCore
CDC ACM/USB2Serial Class Driver
User's Manual**



<http://www.jungo.com>

COPYRIGHT

© Jungo Ltd. 2009 – 2010 All Rights Reserved.

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement. The software may be used, copied or distributed only in accordance with that agreement. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or any means, electronically or mechanically, including photocopying and recording for any purpose without the written permission of Jungo Ltd.

Brand and product names mentioned in this document are trademarks of their respective holders and are used here only for identification purposes.

Contents

Table of Contents	2
List of Figures	6
1 Overview	7
1.1 Introduction	7
1.2 CDC ACM	8
1.2.1 Serial Communication Over USB	8
1.2.2 Use Cases	8
1.2.2.1 Modem Device	9
1.2.2.2 Diagnostics Channel	9
1.2.2.3 Other Devices	9
1.3 Jungo's ACM/USB2Serial Driver	10
1.3.1 Key Features	11
1.3.2 Key Benefits	11
1.3.3 Evaluation	11
2 Getting Started	12
3 Generating a Customized Driver Package using the DriverCore Wizard	13
4 Installing and Uninstalling the Drivers	18
4.1 Installing the Drivers	18
4.1.1 Hardware Requirements	19
4.1.2 Software Requirements	19
4.1.3 Installing the Drivers using the Windows Driver Package Installer (DPIInst)	20
4.1.4 Installing the Drivers using the Windows Plug and Play Manager	20
4.2 Testing the Driver Installation	21
4.3 Uninstalling the Drivers	22

<i>CONTENTS</i>	3
4.3.1 Uninstalling the Drivers from the Device Manager	22
4.3.2 Uninstalling the Drivers Using DPInst	23
4.3.3 Uninstalling the Drivers Using the Windows Add or Remove Programs Wizard	23
5 Distributing the Drivers	24
5.1 Preparing a Driver Installation Package	24
5.2 Digital Driver Signing and Certification	26
5.2.1 Overview	26
5.2.2 WHQL Driver Certification	27
5.2.3 Authenticode Driver Signature	28
6 Advanced Topics	29
6.1 Manually Preparing a Customized Driver Package	29
6.1.1 Adding Control Device Support to a Driver Package	33
6.1.2 Manually Preparing a Build Package	34
6.2 Configuring the ACM Driver Modem Command Set	36
6.3 Power Management	36
6.4 Maintaining State during Device Disconnection	37
6.5 Isolating Child Devices from Plug and Play	37
6.5.1 Introduction	37
6.5.2 Working with the Control Device	39
6.5.3 Installing the Control Device	39
6.5.4 Uninstalling the Control Device	39
7 Accessing the ACM/USB2Serial Device	40
7.1 Overview	40
7.2 Transferring Serial Data Using the Windows API	41
7.3 Transferring Encapsulated Data Using the DriverCore API	42
7.3.1 Windows COM Port Access	42
7.3.2 Driver Modes	42
7.3.3 Single-Application Implementation	43
7.3.4 Dual-Application Implementation	45
A About DriverCore	47
B The Customized Driver Package	48
C The Windows Driver Package Installer (DPInst)	50
D Customizing Driver File Properties via the Resource Files	51
E Manually Customizing the Drivers via the INF Files	52
E.1 Custom DriverCore Driver Parameters	53

<i>CONTENTS</i>	4
E.2 USB Hardware Parameters	53
E.3 Device Hardware Parameters	54
E.3.1 General Parameters	54
E.3.2 CDC ACM Child-Device Parameters	55
E.3.3 Non-Standard-CDC-Device Parameters	56
E.4 Control Device Hardware Parameters	56
F Configuring Device Settings using the Windows Device Manager	57
G Providing CDC ACM Functionality on Windows 2000	58
H Automatic Repair of Modem-Device Symbolic Links	59
I API Reference	60
I.1 Acquiring and Releasing Device Handles	60
I.1.1 Serial-Device Handle	60
I.1.2 Extra-Control Device Handle	61
I.2 DriverCore Common API	62
I.2.1 Acquiring and Releasing Device Handles	62
I.2.1.1 dc_devices_get()	64
I.2.1.2 dc_devices_put()	66
I.2.1.3 dc_device_open()	67
I.2.1.4 dc_device_close()	68
I.3 DriverCore CDC API	69
I.3.1 Acquiring and Releasing Extra-Control Device Handles	69
I.3.1.1 dc_acm_ctrl_handle_get()	70
I.3.1.2 dc_acm_ctrl_handle_put()	71
I.3.2 Encapsulated-Data Transfer Functions	72
I.3.2.1 dc_cdc_encapsulated_send()	73
I.3.2.2 dc_cdc_encapsulated_get()	74
I.3.2.3 dc_cdc_encapsulated_wait()	75
I.4 DriverCore Status Codes API	76
I.4.1 dc_status2str()	76
I.4.2 dc_status_t – DriverCore Status Codes	77
J Troubleshooting	78
J.1 Troubleshooting FAQ	78
J.1.1 My drivers fail to install	78
J.1.2 My composite USB device is not working properly	78
J.1.3 Windows does not recognize the multiple functions of my composite USB device	78
J.1.4 My USB device fails to respond to read and write requests	78

J.1.5	The Windows Device Manager shows an error ("!") for the COM port/modem associated with my device	78
J.1.6	The Windows Device Manager indicates a port conflict for the COM port associated with my device	79
J.1.7	How can I get additional help?	79
J.2	Verifying the Hardware IDs	79
J.3	Troubleshooting a Composite USB Device	80
J.3.1	Checking the Enumeration of a Composite Device	81
J.3.2	Installing the USB Composite Class Driver	82
J.3.3	Verifying the Device Descriptors of a Composite Device	82
J.4	Troubleshooting COM Port Problems	83
J.4.1	COM Port Conflicts – Overview	83
J.4.2	Checking for a COM Port Conflict	84
J.4.3	Reassigning the COM Port for Your Device	84
J.4.3.1	Reassigning the COM Port using the Device Manager	85
J.4.3.2	Reassigning the COM Port using the DriverCore change_port Utility	85
J.5	Debugging the DriverCore Drivers	86
J.5.1	Setting the Debug Level	86
J.5.1.1	Setting the Debug Level in the INF Files	87
J.5.1.2	Setting the Debug Level in the Registry	87
J.5.2	Capturing Debug Information using DebugView	88
J.6	Getting Help	89
K	Contacting Jungo	90

List of Figures

1.1	Jungo’s ACM/USB2Serial Driver Architecture	10
3.1	DriverCore Wizard – Select Your Devices	14
3.2	DriverCore Wizard – Configure the Driver Package	15
3.3	DriverCore Wizard – Advanced Settings	16
3.4	DriverCore Wizard – Choose the Target Folder for the Driver Package	17
6.1	USB Device Stack without a Control Device	37
6.2	Separate USB Device Stacks with a Control Device	38
7.1	Encapsulated-Data Transfer – Single-Application Implementation . .	44
7.2	Encapsulated-Data Transfer – Dual-Application Implementation . . .	46

Chapter 1

Overview

1.1 Introduction

This manual describes Jungo's CDC ACM/USB2Serial Class Driver (**ACM/USB2Serial driver**): its features, how to install it, and how to use it.

Jungo's ACM/USB2Serial driver enables serial applications to communicate with USB devices, by exposing them to the operating system as virtual serial communications ports (**virtual COM ports**) or as virtual serial modems. The ACM/USB2Serial driver leverages your existing serial applications, enabling them to communicate with USB devices as if they were serial devices. The serial applications communicate, without modification, through the operating system serial layers. This eliminates the development costs of making the applications USB-aware.

The driver complies with the **CDC ACM Specification**: the Communication Device Class (**CDC**) Abstract Control Model (**ACM**), as defined in the **CDC 1.1 Specification** (*Universal Serial Bus Class Definitions for Communication Devices, Version 1.1, January 19, 1999* – http://www.usb.org/developers/devclass_docs/usbcdc11.pdf).

1.2 CDC ACM

1.2.1 Serial Communication Over USB

Most manufacturers choose USB connectivity for their peripheral devices. The Universal Serial Bus (**USB**) provides many benefits; it is

- Versatile
- Hot-pluggable
- Plug and Play
- Inexpensive
- Based upon stable, open standards

Computer manufacturers are eliminating legacy serial ports on their products in favor of the more popular USB ports. Modern workstations often lack legacy serial ports, and are unable to directly connect to legacy serial devices. Therefore, device manufacturers are updating even older legacy serial hardware devices to connect via USB. For example, serial modems are being updated with USB connectivity. However, this change creates a barrier for existing applications. A legacy serial application is unable to directly interface with a USB device without modification – a potentially complicated and expensive software project.

To facilitate the integration of serial software with USB hardware, the USB Implementers Forum wrote the CDC ACM Specification, which defines methods for communicating with USB devices using serial communications – controlling the USB devices using V.25ter (**AT**) commands and accessing the USB data streams. A CDC ACM class driver can expose USB communication devices to the operating system as virtual serial communications ports or as virtual serial modems.

1.2.2 Use Cases

Use the CDC ACM class whenever a USB device must be exposed to the operating systems using a legacy serial interface. The following scenarios illustrate some typical examples.

1.2.2.1 Modem Device

A company manufacturers serial modems that provide dial-up access to the Internet or to private computer networks. Customers are upgrading to new computers that lack support for legacy serial devices; the computers only support USB. The existing product line is rapidly becoming obsolete. The manufacturer must update the product line to support USB.

Use a CDC ACM class driver to expose the USB modem to the operating system as a standard serial modem connected to a COM port (virtual serial modem). This enables modem configuration via the operating system (for example, via the Windows Control Panel) and retains the ability to use AT modem commands. Once the class driver has been installed, serial applications communicate with the USB device, without modification.

1.2.2.2 Diagnostics Channel

A device provides services for diagnostics, scheduling, or clock synchronization. Applications access the device using legacy serial APIs. Even though the device is highly successful, other devices are appearing on the market that use the cheaper, now-standard USB interface. To remain a market-leader, the manufacturer must update the product line to support USB.

Use a CDC ACM class driver to expose the serial-data stream to the operating system as a standard COM port (virtual serial communications port). This enables serial-port configuration via the operating system (for example, via the Windows Control Panel). Once the class driver has been installed, serial applications communicate with the USB device, without modification.

1.2.2.3 Other Devices

Use a CDC ACM class driver to add USB support to any device that uses serial-data communications:

- Point of service equipment, such as barcode scanners and dedicated printers
- Test and measurement devices
- GPS devices that use a serial interface for information updates, such as maps, NMEA waypoints, and other relevant data
- Applications with serial port control of medical devices, industrial machinery, etc.

1.3 Jungo's ACM/USB2Serial Driver

Using the CDC ACM Specification as a starting point, Jungo built an effective solution to the problem of using legacy serial devices over USB. Jungo's CDC ACM/USB2Serial Class Driver (**ACM/USB2Serial driver**) provides the missing middle layer – communicating with the USB host stack, while exposing legacy serial connectivity to the user and the software applications. With no modification, a legacy serial application can use the driver-provided virtual COM port to communicate with a USB device.

Figure 1.1 illustrates Jungo's ACM/USB2Serial driver solution.

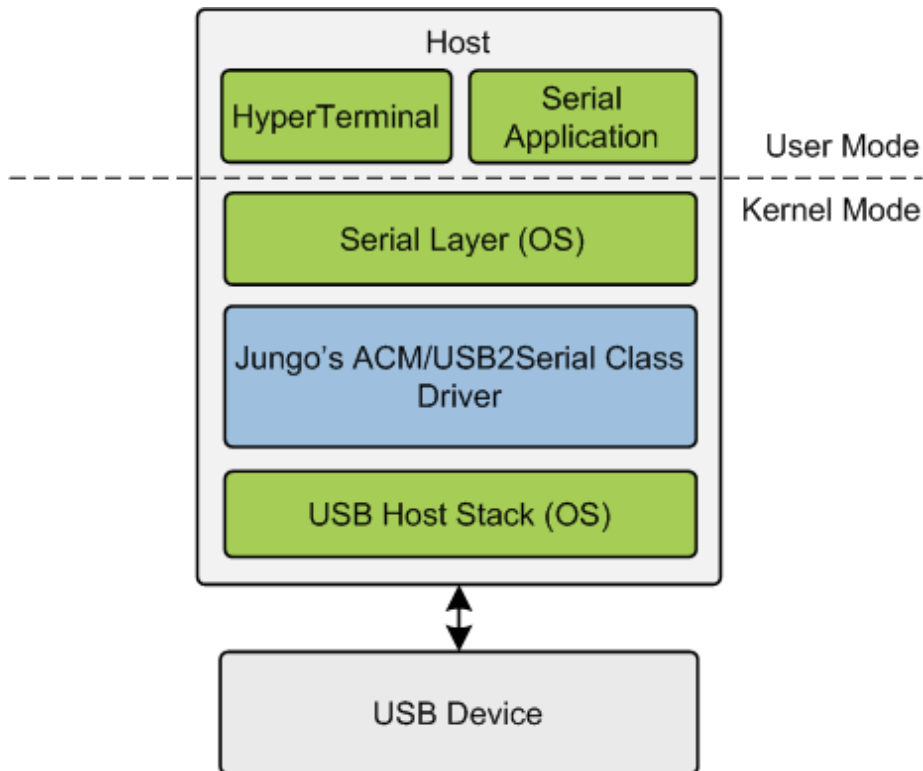


Figure 1.1: Jungo's ACM/USB2Serial Driver Architecture

i The ACM/USB2Serial driver is part of Jungo's DriverCore product line [A].

1.3.1 Key Features

- Complies with the CDC ACM Specification
- Supports
 - Proprietary USB-to-serial designs
 - Enumeration as a virtual COM port
 - Enumeration as a virtual serial modem device
 - Serial events, such as TX empty, RX ready, Received Line Signal Detect (RLSD), and RING
 - Serial-data transfer
 - Encapsulated-data transfer
 - Automatic repair of modem-device symbolic links [\[H\]](#)
 - USB power management, including selective suspend
 - Synchronous I/O and overlapped I/O
 - Simultaneous connection of multiple USB devices
 - Composite USB devices with multiple CDC ACM functions

1.3.2 Key Benefits

- Turnkey solution: eliminates substantial development time and costs
- Hardware- and software-independent solution
- Modular driver
- Available as source code or as object code
- Cross-platform across the supported operating systems and CPU architectures, which include x86, x64, and ARM
- Intuitive API, written in C
- Extensive documentation
- Comprehensive technical support

1.3.3 Evaluation

Jungo offers a limited evaluation license for the ACM/USB2Serial driver. For more information, contact Jungo [\[K\]](#).

Chapter 2

Getting Started

To create and use your own customized DriverCore ACM/USB2Serial driver, perform these steps:

1. Generate a customized driver package using the DriverCore wizard [3] (recommended), or prepare the package manually [6.1]
2. Optionally, perform additional manual driver customization
3. Install the drivers [4.1]
4. Test the driver installation [4.2]
5. Distribute your drivers [5]

Chapter 3

Generating a Customized Driver Package using the DriverCore Wizard

The DriverCore wizard (**dc_wizard**) is a GUI application that provides a simple and reliable way to generate a customized driver package for your USB devices.

The wizard guides you through the following steps:

1. Selecting the target devices that the driver will handle
2. Customizing the vendor branding information of the drivers (e.g., manufacturer name and driver name)
3. Configuring advanced driver settings (optional)
4. Generating your customized driver package
5. Installing the drivers

NOTE

The wizard can generate a single driver package that supports multiple USB devices.

To generate a customized driver package

1. Start the DriverCore wizard. The wizard application file (e.g., **dc_wizard.exe** on Windows) is located in the DriverCore root directory. The wizard's welcome screen is displayed.
2. Click **Next >** to display the device selection window.

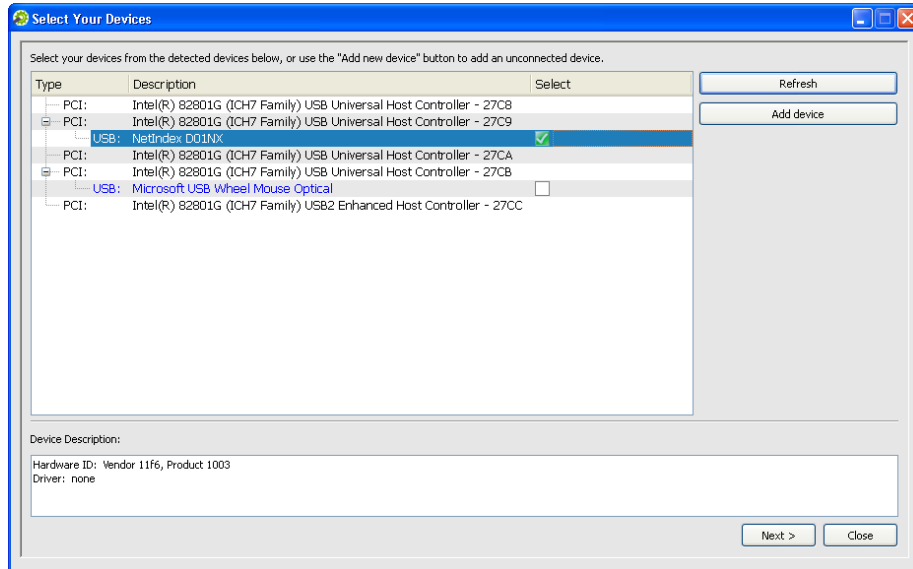


Figure 3.1: DriverCore Wizard – Select Your Devices

3. Select the checkboxes of the USB devices to include in the driver package.

i To add a device that is not currently connected to the host, either connect the device and click **Refresh**, or click **Add device** and enter the device details.

4. Click **Next >** to display the driver package configuration window (see Figure 3.2 below).
 - The upper area of the window lists the devices that you selected in the previous step. Each row contains text fields that enable you to customize the configuration information for a specific device.
 - The lower area of the window contains text fields that enable you to customize configuration information that applies to all of the devices.

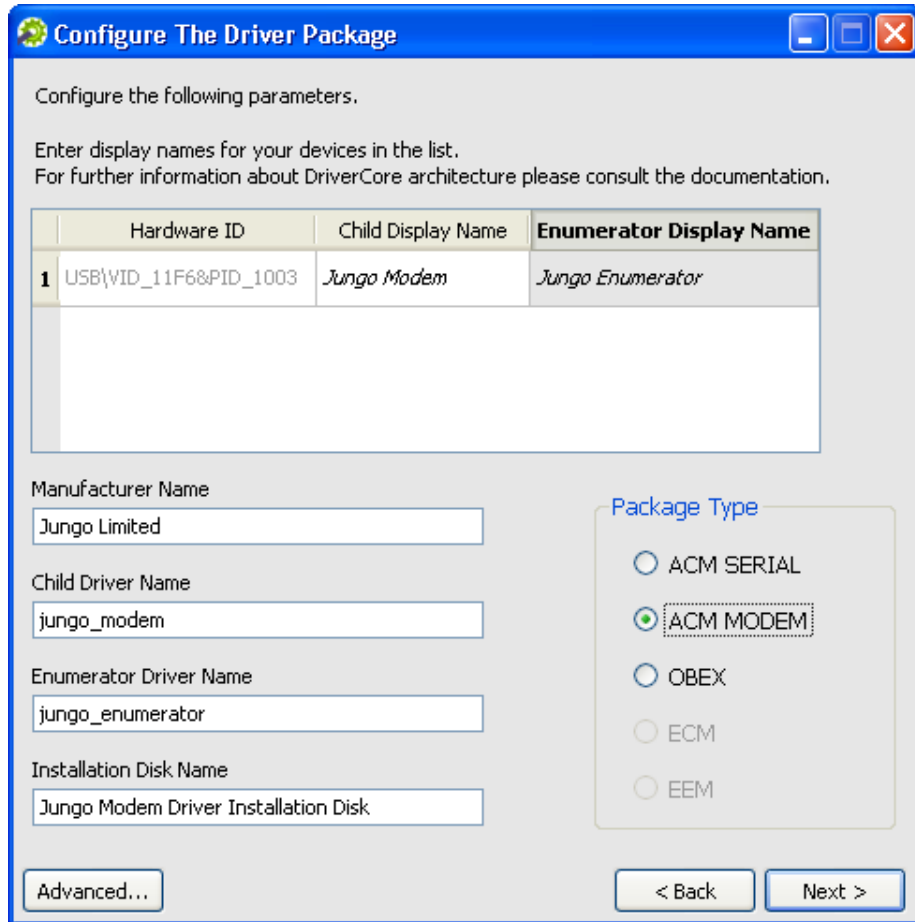


Figure 3.2: DriverCore Wizard – Configure the Driver Package

5. Enter **display names** for the enumerator and child logical devices.
Choose display names that will help users identify your devices within the operating system device lists.
6. Enter your custom text in each of the remaining fields.
The **driver name** fields determine the file names of the drivers.
Note: It is important to use unique company-/product-specific names, to ensure that there is no name conflict with other drivers in the system.
7. In the **Package Type** area, choose the device category that matches your target devices.

8. To configure non-CDC-compliant drivers or disable zero-length packet
 - (a) Click **Advanced...** to display the **Advanced Settings** window [Figure 3.3].
 - (b) Configure the desired parameters, then click **OK**.
The **Advanced Settings** window closes.

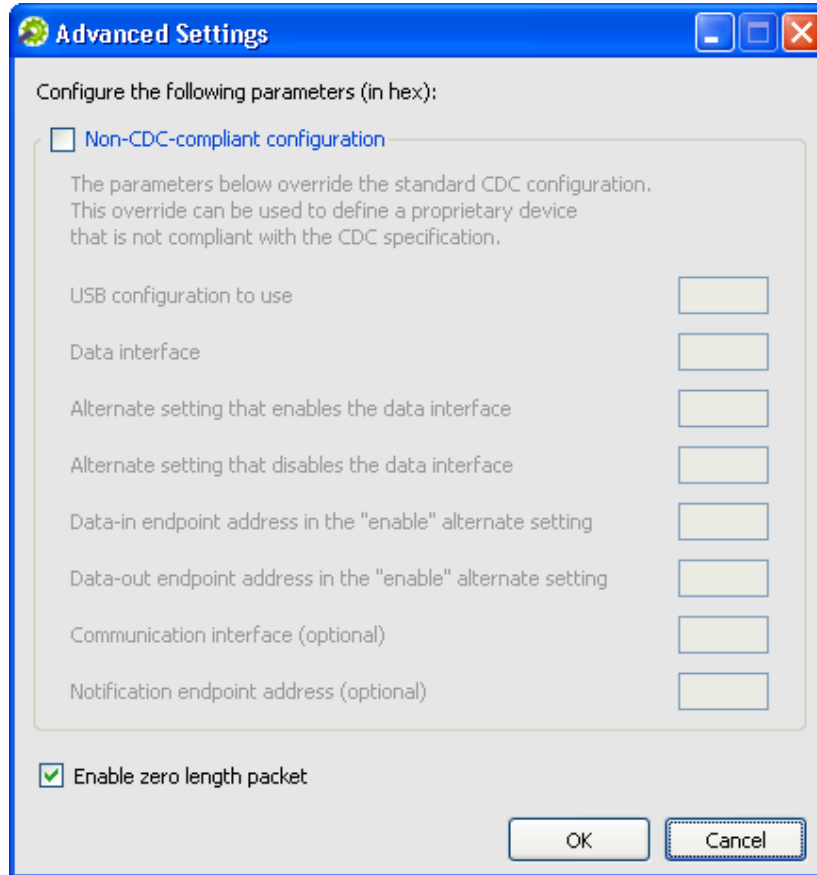


Figure 3.3: DriverCore Wizard – Advanced Settings

- Click *Next* > to display a browse dialogue; select the target directory (folder) in which to store the generated driver package, and click *OK*.

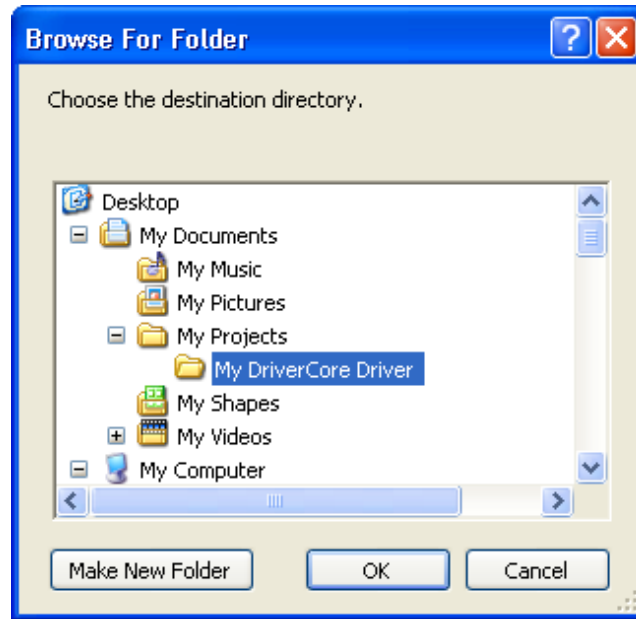



Figure 3.4: DriverCore Wizard – Choose the Target Folder for the Driver Package

- The wizard attempts to generate the driver package, and displays the result. Upon success, you are given the option to install the drivers. If you select *Yes*, the wizard initiates driver installation; follow the instructions in the device driver installation wizard to complete the installation.

 Appendix B describes the contents of the generated driver package.

NOTE

For an **ACM modem device**, before installing the drivers you may need to modify the generated modem child driver INF file, to configure the modem command set for your specific device, as explained in section 6.2 of the manual.

Chapter 4

Installing and Uninstalling the Drivers

4.1 Installing the Drivers

To provide driver services to the operating system, the drivers must first be installed on the target machine.

NOTE

If you decide to use the control device mode, you must install the control device before installing the driver package. For more information, see *Isolating Child Devices from Plug and Play* [6.5].

There are two methods for installing DriverCore drivers on a Windows machine:

1. Using the Windows Driver Package Installer (**DPInst**) [4.1.3] (recommended).

Note: The DriverCore wizard uses DPInst for the optional driver installation (see Chapter 3, step #10).

2. Using the Windows Plug and Play Manager [4.1.4].

NOTE

DPIInst has several advantages over the Windows Plug and Play Manager installation:

- Minimizes user-interaction
- Can be used from your own software installer
- Works regardless of whether the USB device is connected to the computer
- Enables automated uninstallation using the Windows Add or Remove Programs Wizard [4.3.3]

NOTE

- In this documentation, ”**driver installation package**” refers to the package that contains the driver installation files. This can be either your customized DriverCore driver package [B] or a package created specifically for installation [5.1].
- If your device has not been WHQL-certified together with the DriverCore drivers, and/or if the drivers have not been digitally signed, warning dialogue boxes will appear during installation; you may click *Continue Anyway* (except for 64-bit editions of Windows Vista and higher). To enable a smooth installation experience, WHQL-certify your devices with the DriverCore drivers [5.2].

4.1.1 Hardware Requirements

DriverCore supports Windows systems using the following CPU types:

- Any x86 32-bit processor
- Any x64 64-bit processor (AMD64 or Intel 64)

4.1.2 Software Requirements

DriverCore supports the following Windows versions:

- Windows XP (SP2 or higher)
- Windows 2000 (SP4 or higher) – for compatible devices [G]
- Windows Server 2003 (SP1 or higher)
- Windows Vista (SP1 or higher)

Before installing the drivers, update Windows to the specified service pack level.

4.1.3 Installing the Drivers using the Windows Driver Package Installer (DPInst)

NOTE

Before performing the installation procedure described in this section, verify that the **dpinst.xml** file in your driver installation package lists all the relevant INF files – namely, the enumerator and child driver INF files, and the Extra-Control Device INF file – if you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2].

If you have renamed the default files, you will need to edit **dpinst.xml** to include the correct file names (see information in section 6.1, step #9).

To install the drivers using the Windows Driver Package Installer (**DPInst**) [C]

1. Navigate to the directory containing the driver installation package for the target CPU (e.g., **x86** or **x64**).
2. From this directory, start the Windows Driver Package Installer – **DPInst.exe**. The Windows Device Driver Installation Wizard opens.
3. Confirm installation of the drivers and follow the instructions in the wizard to complete the installation.



DPInst has several advantages over the Windows Plug and Play Manager installation [4.1.4], as explained above [4.1].

4.1.4 Installing the Drivers using the Windows Plug and Play Manager

NOTE

As explained above, it is recommended to use the DPInst installation method [4.1.3] instead of the method described in this section.

To install the drivers using the Windows Plug and Play Manager

1. Connect your USB device to a USB port on the computer.
After a few moments, the Windows Plug and Play Manager automatically detects the new hardware and opens the Found New Hardware Wizard.
The wizard asks whether you would like to use Windows Update.
2. Select **No, not this time**, then click **Next >**.
The wizard asks what you want to do.

3. Select ***Install from a list or a specific location (Advanced)***, then click ***Next >***.
The wizard prompts for search and installation options.
4. Click ***Browse***.
The *Browse for folder* dialogue box opens.
5. Browse to the directory (folder) containing the driver installation package for the target CPU (e.g., **x86** or **x64**).
6. Select the **driver** subdirectory, then click ***OK***.
The *Browse for folder* dialogue box closes.
7. Click ***Next >***.
8. Confirm installation of the drivers and follow the instructions of the wizard, to complete installation of the enumerator driver.
The wizard indicates that the driver is installed.
9. Click ***Finish***
The Plug and Play Manager again detects new hardware and opens the Found New Hardware Wizard.
The wizard asks whether you would like to use Windows Update.
10. Repeat this procedure, starting from step #2, to install the child driver INF file and any additional required INF files, such as the Extra-Control Device INF file [7.3.2].

4.2 Testing the Driver Installation

If the drivers are properly installed for your USB device, the Windows Device Manager will display your device in the device tree.

To verify that the drivers are installed, perform these steps:

1. Connect your USB device to the computer.
2. Open the Device Manager (**devmgmt.msc**).
3. From the ***View*** menu, select ***By Type***.
4. Locate your device within the device tree:
 - ACM modem devices are displayed under ***Modems***.
 - ACM serial devices and OBEX devices are displayed under ***Ports***.

NOTE

If your device is not listed in the Device Manager – i.e., the drivers failed to install – refer to the *Troubleshooting* appendix [J.1.1].

4.3 Uninstalling the Drivers

If the driver services are no longer needed, the drivers can be uninstalled from the target machine.

There are several methods for uninstalling the DriverCore drivers on Windows:

- Using the Windows Device Manager [4.3.1].
- Using DPNst command-line uninstallation [4.3.2].
- Using the Windows Add or Remove Programs Wizard.

NOTE

The Windows Add or Remove Programs Wizard method can only be used to uninstall drivers that were installed using DPNst [4.1.3].

NOTE

If the control device is installed, after uninstalling the driver package you must uninstall the control device. For more information, see *Isolating Child Devices from Plug and Play* [6.5].

- i** The level of clean-up performed by the uninstallation may differ, depending on the uninstallation method and the version of the operating system. We recommend using the Windows Add or Remove Programs Wizard [4.3.3] or DPNst [4.3.2] to uninstall the DriverCore drivers.

4.3.1 Uninstalling the Drivers from the Device Manager

To uninstall a driver using the Windows Device Manager, repeat the following sequence for each INF file that you installed for the driver:

1. Open the Device Manager (**devmgmt.msc**).
2. In the device tree, locate the device node for the installed INF file.
3. Right-click the device node and select **Uninstall**.

NOTE

On some versions of Windows, this method leaves traces of the driver in the system.

4.3.2 Uninstalling the Drivers Using DPInst

To uninstall your drivers using the Windows Driver Package Installer (DPInst) [C]

1. Open a command prompt.
2. Change directory to your driver installation package directory. For example:
`cd C:\DriverCore\x86`
3. For each INF file that you installed using DPInst, run **DPInst.exe** with the uninstall option `- /u`:
`DPInst.exe /u <inf-file-path>`

Note: The DriverCore driver INF files are located in the **driver** directory of your driver installation package; therefore, the INF path should normally be **driver/<INF file name>**.

For example, to uninstall the enumerator driver named **my_enum**, run
`DPInst.exe /u driver/my_enum.inf`

If you installed the drivers using DPInst [4.1.3], you can find the list of installed INF files in the **dpinst.xml** file that was used in the installation.

NOTE

You may also wish to use the `/d` option to delete the binary files that were copied to the system as part of the driver installation:

```
DPInst.exe /u <inf-file-path> /d
```

 This method is not limited to drivers that were installed using DPInst.

4.3.3 Uninstalling the Drivers Using the Windows Add or Remove Programs Wizard

If the drivers were installed using the Windows Driver Package Installer (DPInst) [4.1.3], they can be uninstalled using the Windows Add or Remove Programs Wizard:

1. Close any open applications that access your USB device.
2. Start the Windows Add or Remove Programs Wizard.
3. In the *Currently installed programs* list, locate the row containing the software to be uninstalled. The drivers should be listed as:
Windows Driver Package - <Manufacturer> (<Driver name>) USB.
4. Select the row, then click **Change/Remove**.
The *Uninstall Driver Package* dialog box opens, requesting confirmation.
5. Click **Yes**.
The Windows Driver Package Installer (DPInst) [C] uninstalls the drivers.

Chapter 5

Distributing the Drivers

To distribute your DriverCore drivers

1. Prepare a driver installation package, which contains the driver files and all other files required for installing the drivers on target machines [5.1].
2. Install the drivers on target machines.
The supported driver installation methods are described in section 4.1.

TIP

For a smooth installation experience, it is recommended that you WHQL-certify and/or digitally sign your drivers before distributing them [5.2].

5.1 Preparing a Driver Installation Package

Before distributing your drivers, you need to prepare a driver installation package. Your customized driver package [B] contains the files that are required for driver installation, as well as additional files. This section guides you through the process of creating a driver installation package that includes only the required files.

NOTE

In the following instructions

- The **”driver package directory”** is the customized driver package directory for the target platform (e.g., **x86** or **x64**), as generated by the DriverCore wizard [3] or prepared manually [6.1].
- Replace **<enum>** and **<child>** instances with the names of your selected enumerator driver and child driver, respectively. For example, if the name of your enumerator driver is **jungo_enum**, replace **<enum>.sys** with **jungo_enum.sys**. In addition, replace **<extra_ctrl>** instances with the name of the Extra-Control Device [7.3.2].

To create a custom driver installation package

1. Create a new installation directory (**”the installation directory”**).
2. To support installation of the drivers using DPInst [4.1.3], copy the following files from the driver package directory to the installation directory:
 - **DPInst.exe** – the Microsoft Windows Driver Package Installer (DPInst) [C]
 - **difxapi.dll** – the Microsoft Driver Install Frameworks API (DIFxAPI) DLL, required by DPInst
 - **dpinst.xml** – DPInst descriptor (configuration) file
3. In the installation directory, create a **driver** subdirectory, and copy to this subdirectory the following files from the driver package **driver** subdirectory:
 - **WdfCoInstaller01007.dll** – the Microsoft Windows Driver Foundation (WDF) co-installer DLL, which is required for the operation of the DriverCore drivers
 - **<enum>.sys** (default: **dc_enum.sys**) – the enumerator driver, which provides the interface between the USB and the child driver
 - **<enum>.inf** (default: **dc_enum.inf**) – an INF file for installing the enumerator driver
 - **<child>.sys** (default: **cdc_acm.sys**) – the child driver
 - **<child>.inf** (default: **cdc_acm_modem.inf** or **cdc_acm_obex.inf**) – an INF file for installing the child driver

The following **driver** directory files are only required if you have selected to use the control device mode – see *Isolating Child Devices from Plug and Play* [6.5]. Otherwise, they need not even be included in your customized driver package:

- **<enum>_control.inf** (default: **dc_enum_control.inf**) – an INF file for creating the control device
- **wdreg.exe** – a utility for installing/uninstalling the control device INF
- **wdreg_gui.exe** – a graphical version of **wdreg.exe**

The following **driver** directory files are only required if you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2]; otherwise, they need not even be included in your customized driver package:

- **<extra_ctrl>.inf** (default: **cdc_acm_ser_ctrl.inf**) – an INF file for creating an Extra-Control Device [7.3.2]

5.2 Digital Driver Signing and Certification

5.2.1 Overview

Before distributing a Windows driver, you can certify and/or digitally sign it; either submit the driver to the Microsoft Windows Logo Program, for WHQL certification and signature [5.2.2], or have the driver Authenticode signed [5.2.3].

Some Windows operating systems, such as Windows XP and below, do not require installed drivers to be digitally signed or certified. There are, however, advantages to getting the driver digitally signed or fully certified, including the following:

- Driver installation on systems where installing unsigned drivers has been blocked
- Avoiding warnings during driver installation
- Avoiding the need to re-install the driver, when moving the device to a different USB port
- Full pre-installation of INF files on Windows XP and higher

64-bit versions of Windows Vista and higher, e.g., Windows Server 2008, require Kernel-Mode Code Signing (KMCS) of software that loads in kernel mode. Therefore, DriverCore-based drivers must be WHQL certified or distributed together with an Authenticode-signed catalog file, to permit installation under these versions of Windows.

TIP

During driver development, you can configure Windows to temporarily allow the installation of unsigned drivers.

For more information about digital driver signing and certification, see:

- Driver Signing Requirements for Windows:
<http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspc>.
- The *Introduction to Code Signing* topic in the Microsoft Developer Network (MSDN) documentation.
- Digital Signatures for Kernel Modules on Systems Running Windows Vista:
<http://www.microsoft.com/whdc/winlogo/drvsign/kmsigning.mspc>.
This white paper contains information about kernel-mode code signing, test signing, and disabling signature enforcement during development.

5.2.2 WHQL Driver Certification

The Microsoft Windows Logo Program defines procedures for submitting hardware and software modules, including drivers, for Microsoft quality assurance tests. Passing the tests qualifies the hardware/software for Microsoft certification, which verifies both the driver provider's authenticity and the driver's safety and functionality.

Device drivers should be submitted for certification together with the hardware that they drive. The driver and hardware are submitted to Microsoft's Windows Hardware Quality Labs (**WHQL**) testing in order to receive digital signature and certification. This procedure verifies both the driver's provider and its behavior.

To enable a smooth installation experience, WHQL-certify your devices together with the DriverCore drivers. For information about Jungo's WHQL services, please contact Jungo [K].

For detailed information regarding the WHQL certification process, refer to the following Microsoft web pages:

- WHQL home page:
<http://www.microsoft.com/whdc/whql/default.mspc>
- WHQL Policies page:
<http://www.microsoft.com/whdc/whql/policies/default.mspc>

5.2.3 Authenticode Driver Signature

The Microsoft Authenticode mechanism verifies the authenticity of the driver provider using a digitally signed catalog file, containing information about the driver developers and their code. By providing the catalog file together with the software, the publisher informs the users of the driver that the publisher is participating in an infrastructure of trusted entities. The Authenticode signature does not, however, guarantee the code's safety or functionality.

Chapter 6

Advanced Topics

6.1 Manually Preparing a Customized Driver Package

This section describes how to manually prepare a customized Windows driver package for your USB device(s), including

1. How to create a new customized driver package from the template files
2. How to configure the driver package by editing the INF files; this includes
 - (a) Defining the devices' hardware IDs
 - (b) Customizing the vendor branding information of the drivers (e.g., manufacturer name and driver name)

Note: For additional information regarding manual configuration of driver parameters via the INF files, refer to appendix E.

 Appendix B describes the contents of DriverCore driver packages.

TIP

It is easier to generate a customized driver package automatically, using the DriverCore wizard, as outlined in Chapter 3.

To manually prepare a customized driver package, follow these steps:

1. Create a driver package directory for the target CPU; for example, **x86** for installation on 32-bit computers, or **x64** for installation on 64-bit computers.

NOTE

In the following documentation, replace **<pkg-target-dir>** instances with the full path to the target driver package directory that you created, for example **C:\Jungo_Modem_Driver\x86**.

2. Create a **driver** directory in your new driver package directory:
<pkg-target-dir>\driver.

NOTE

In the following documentation, replace **<driver-target-dir>** instances with the full path to this **driver** directory, for example **C:\Jungo_Modem_Driver\x86\driver**.

3. From the appropriate DriverCore source CPU directory, e.g., **x86** or **x64**, copy **difxapi.dll** and **DPInst.exe** to **<pkg-target-dir>**.
4. From the DriverCore **template** directory, copy **dpinst_template.xml** to **<pkg-target-dir>**.
Rename the copy to **dpinst.xml**.
5. From the appropriate DriverCore **driver** directory, e.g., **x86\driver** or **x64\driver**, copy the following files to **<driver-target-dir>**:
 - **WdfCoInstaller01007.dll**
 - **dc_enum.sys**
 - **dc_enum.pdb**
 - **cdc_acm.sys**
 - **cdc_acm.pdb**

The following files are required only if you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2]:

- **cdc_acm_ser_ctrl.inf**

6. From the DriverCore **template** directory, copy the following files to **<driver-target-dir>**:
 - The enumerator INF file template: **dc_enum_template.inf**
 - The appropriate child INF file template for your target child device:
 - **cdc_acm_modem_template.inf** – for ACM modems
 - **cdc_acm_obex_template.inf** – for non-modem ACM or OBEX devices

7. Choose unique enumerator and child driver names. If you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2], you can also choose a unique Extra-Control Device name. The selected names will be used to name the files associated with the renamed modules.

The documentation uses the following notation to refer to your selected names:

- **<enum>** – the enumerator driver name, e.g., **jungo_enum**
- **<child>** – the child driver name, e.g., **jungo_modem**
- **<extra_ctrl>** – the Extra-Control Device name, e.g., **jungo_extra_ctrl_dev**

Note: These notations do *not* include file extensions (such as ".inf").

8. In **<driver-target-dir>**, rename the following files:

- **dc_enum.sys** to **<enum>.sys**
- **dc_enum.pdb** to **<enum>.pdb**
- **cdc_acm.sys** to **<child>.sys**
- **cdc_acm.pdb** to **<child>.pdb**
- **dc_enum_template.inf** to **<enum>.inf**
- The template child INF file (e.g., **cdc_acm_modem_template.inf**) to **<child>.inf**

If you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2], rename the following files as well:

- **cdc_acm_ser_ctrl.inf** to **<extra_ctrl>.inf**

9. In the **<pkg-target-dir>** directory, edit **dpinst.xml**:

- Replace **ENUM_DRIVER_NAME** with **<enum>**.
- Replace **CHILD_DRIVER_NAME** with **<child>**.
- If you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2], duplicate the line that refers to the child driver INF file, and replace the INF file name with the name of your Extra-Control Device INF file:

```
<package path=". \driver\<extra_ctrl>.inf" />
```

For example, for the default Extra-Control Device INF file name –

cdc_acm_ser_ctrl.inf – add the following line:

```
<package path=". \driver\cdc_acm_ser_ctrl.inf" />
```

10. Edit the enumerator INF file – **<enum>.inf** – and the child INF file – **<child>.inf**:

- (a) Replace all instances of the following strings:
- **MANUFACTURER** with your company name, for example **Jungo**
 - **INSTALLATION_DISK** with your installation disk name, for example **Jungo_Modem_Installation**
 - **DRIVER_NAME** with your selected driver name:
 - In **<enum>.inf**, replace **DRIVER_NAME** with **<enum>**.
 - In **<child>.inf**, replace **DRIVER_NAME** with **<child>**.

(b) Replace the **DriverVer** line, in both files, with the equivalent line from the DriverCore enumerator INF file – **dc_enum.inf**, found in the relevant DriverCore **driver** directory for the target CPU (e.g., **x86\driver** or **x64\driver**).

(c) Modify the hardware ID definitions in the **[DeviceList]**, **[DeviceList.NTx86]**, and **[DeviceList.NTamd64]** sections.

The format of the hardware IDs in the template INF files is **<device type>\VID_XXXX&PID_YYYY&MI_ZZ**.

Modify the hardware ID definitions to fit your device:

- Replace **XXXX** with the **vendor ID**.
- Replace **YYYY** with the **product ID**.
- For a composite device, replace **ZZ** with the **interface number** to be associated with the driver.
For a non-composite device, delete the interface number section at the end of the line: **&MI_ZZ**.

The **<device type>** portion of the hardware ID is already defined in the template INF files: it is **USB** in the enumerator INF file, and the appropriate type for your device in the child INF file (e.g., **USBCDCACM**); do *not* modify the device type definitions in the files.

For example, to install the enumerator driver for interface 3 of a composite device with vendor ID 0x1111 and product ID 0x2222, edit the hardware ID in **<enum>.inf** to **USB\VID_1111&PID_2222&MI_3**. For a similar but non-composite device, omit the **&MI_3** portion: **USB\VID_1111&PID_2222**.

NOTE

Each hardware ID line has two instances of the hardware ID:
`%<hardware ID>.DeviceDesc%=DevInstall,<hardware ID>.`
 Be sure to change both hardware ID definitions in each line.

Add similar hardware ID lines for each additional device or composite-device interface that you want to associate with the driver.

- (d) Modify the hardware description lines in the **[Strings]** section –
`<device type>\<hardware ID>#.DeviceDesc = "DISPLAY_NAME"`:
- i. Modify the hardware ID definitions, as explained in step #10c.
 - ii. Replace each **DISPLAY_NAME** device description (**DeviceDesc**) with the desired display name.
 Choose meaningful display names that will help users identify your devices within the operating system device lists.

For example, to identify your ACM modem child device (vendor ID 0x1111, product ID 0x2222) as **Jungo Modem**, edit the line to:
`USB CDCACM\VID_1111&PID_2222#.DeviceDesc = "Jungo Modem"`.

NOTE

You can also modify any of the other strings in the **[Strings]** section.
 If you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2], you can also modify the strings in the Extra-Control Device INF file (`<extra_ctrl>.inf`).

- (e) Delete all `##` occurrences in the files.

6.1.1 Adding Control Device Support to a Driver Package

If your DriverCore distribution includes the optional control device files [6.5.2], you can add control device mode-support to your driver packages.

NOTE

The control device feature is not included in the default DriverCore distribution. Look for `dc_enum_control.inf` in the DriverCore **driver** directory. If it is not there, you can contact Jungo sales [K], to obtain the control device files at no additional charge.

To add control device mode-support to a driver package

1. Copy the following files from the DriverCore **driver** directory to **<pkg-target-dir>**:
 - **wdreg.exe**
 - **wdreg_gui.exe**
 - **dc_enum_control.inf**
2. In **<pkg-target-dir>**, rename **dc_enum_control.inf** to **<enum>_control.inf**.
3. In **<pkg-target-dir>**, open **<enum>_control.inf** in an editor:
 - Replace all occurrences of **dc_enum** with **<enum>**.
 - Edit the strings in the **[Strings]** section. Replace "DC control device" with the desired display name. Choose a meaningful display name that will help users identify your device within the operating system device lists.

6.1.2 Manually Preparing a Build Package

If you select to modify the properties of your driver files, you need to rebuild the drivers, as outlined in Appendix D. To add the required build files to your customized driver package, follow these steps:

NOTE

As explained above [7], in this manual, **<enum>** represents your selected enumerator driver name, and **<child>** represents your selected child driver name.

1. Create **enum_sys** and **child_sys** directories in your **<driver-target-dir>** directory.
2. Copy the required files from the DriverCore **template** directory to the new directories:
 - (a) Copy the following files to both the **enum_sys** and **child_sys** directories:
 - **rc_template.rc**
 - **sources**
 - **makefile**
 - **kernel_util.lib**
 - (b) Copy **dc_enum.lib** to the **enum_sys** directory.
You can rename this file as you wish; for example, rename it to use your enumerator driver name – **<enum>.lib**.

- (c) Copy **cdc_acm.lib** to the **child_sys** directory.
You can rename this file as you wish; for example, rename it to use your child driver name – **<child>.lib**.
3. Rename the copies of the **rc_template.rc** resource file, using the respective driver names:
 - Rename the file in the **enum_sys** directory to **<enum>.rc**.
 - Rename the file in the **child_sys** directory to **<child>.rc**.
4. Edit the renamed resource files – **<enum>.rc** and **<child>.rc** – as follows:
 - Replace **DRIVER_NAME** instances with your selected driver name: **<enum>** in **<enum>.rc**, and **<child>** in **<child>.rc**.
 - Modify the string definitions in the files, as you see fit. For example, in the **VER_COMPANYNAME_STR** string definition, replace **Jungo** with your company's name.
5. Modify the copy of the **sources** file in the **enum_sys** directory:
 - Replace **DRIVER_NAME** instances with your selected enumerator driver name – **<enum>**.
 - Replace **LIB_FILE_NAME** with your selected enumerator driver library name – e.g., **<enum>** (see step #2b).
 - Delete all **\$DDK_LIB_PATH** lines, *except* for the line containing **ntstrsafe.lib**.
6. Modify the copy of the **sources** file in the **child_sys** directory:
 - Replace **DRIVER_NAME** instances with your selected child driver name – **<child>**.
 - Replace **LIB_FILE_NAME** with your selected child driver library name – e.g., **<child>** (see step #2b).
 - Delete all **\$DDK_LIB_PATH** lines, *except* for the lines containing **ntstrsafe.lib** or **wdmsec.lib**.

6.2 Configuring the ACM Driver Modem Command Set

Because many modem command sets are proprietary, the ACM modem driver must be configured for the specific command set of your modem. This configuration is done via the modem child driver INF file. The provided DriverCore version of this INF file is configured for the MicroLink 56K USB modem; for other modems, before installing the modem child driver, edit its INF file to configure the modem command set for your specific modem device.

To configure the modem command set:

1. Open the child driver INF file, using a text editor.
2. Locate the modem-specific parameters (e.g., initialization, settings, and responses). These parameters are located within the **Responses** section.
3. Configure the parameters to use the command set of your modem.

6.3 Power Management

DriverCore fully supports Windows power management methods, including selective suspend. Selective suspend provides the ability to power down and later resume an idle USB device. The idle device is suspended without affecting other devices that are connected to the same hub or, in the case of a multifunction device, without affecting the other functions in the device. When all of the devices or functions have been suspended, the entire hub or multifunction device can be powered down.

When the **IdleTimeout** INF/registry parameter is set [E.1], DriverCore detects when the child driver is idle, and enables selective suspend by suspending the enumerator device.

Selective suspend requires the following prerequisites:

- The host operating system must be running Windows XP or higher.
- The device must support USB remote wakeup.

6.4 Maintaining State during Device Disconnection

Typically, when a USB device is disconnected, the device driver's state is cleared. This is the default DriverCore behavior – when the enumerator (bus) device is unloaded, DriverCore automatically unloads the child device, clearing the driver state.

For some applications, it may be desirable to retain driver state, in case the device is later reconnected. For example, a terminal session could be maintained even though the USB modem was temporarily disconnected from the computer. DriverCore's supports this special behavior – when the `CtrlDevHoldChildIfInUse` INF/registry parameter is set [E.3.2], DriverCore delays unloading the child device until all handles to it are closed.

NOTE

This feature requires control device mode, to isolate the child device from Plug and Play [6.5].

6.5 Isolating Child Devices from Plug and Play

6.5.1 Introduction

The enumerator driver (bus driver) is loaded by the physical device object (PDO) which handles the interrupt request packets (IRPs) for the physical USB device that is connected to the USB bus (USB controller). The child drivers (function drivers) are loaded by the functional device objects (FDOs) which handle IRPs for the USB functions that the device can handle.

In the typical USB device-driver stack, the function drivers (child devices) are located under the bus driver (enumerator device) in the stack. This is also the DriverCore default configuration. The following figure uses a section from the Windows Device Manager to illustrate this configuration.



Figure 6.1: USB Device Stack without a Control Device

For certain types of applications, it is desirable to isolate the FDOs from Plug and Play, which isolates the PDOs from the FDOs. Plug and Play isolation enables

- Driver control over power management – required for the ECM Driver to support selective suspend.
- Driver control over the loading and unloading of the child driver. This enables the driver to retain state, even when the device is disconnected from the USB. For example, when used with the ACM/USB2Serial driver, a terminal session can be maintained even though the USB modem is disconnected temporarily from the computer [6.4].

DriverCore supports a means of isolating the FDOs from the PDO: the creation of a control device. The control device is created manually, as a *virtual device*, at the root level of the Windows device stack. The FDOs (child devices) are located under the control device, isolating them from physical bus events and from Plug and Play. This is called *control device mode*. The following figure uses a section from the Windows Device Manager to illustrate this configuration.

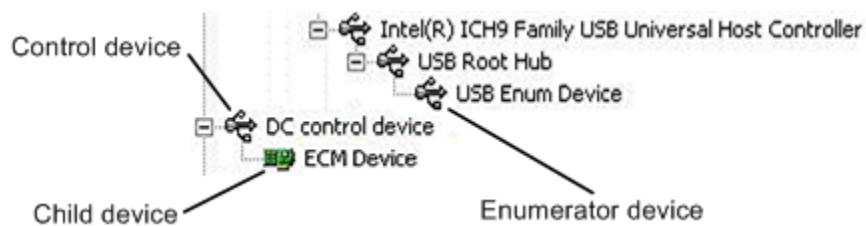


Figure 6.2: Separate USB Device Stacks with a Control Device

When DriverCore is in control device mode, the control device proxies API requests between the child devices and the enumerator device, for example:

1. The child device calls a DriverCore API function.
2. The child device passes the request to the control device.
3. The control device verifies that the enumerator device exists.
4. If the enumerator device exists, the control device passes the request to the enumerator driver.

6.5.2 Working with the Control Device

To enable the DriverCore control device mode, install the control device [6.5.3]. To disable the DriverCore control device mode, uninstall the control device [6.5.4].

NOTE

The control device feature is not automatically included in DriverCore driver packages. For more information, see *Adding Control Device Support to a Driver Package* [6.1.1].

6.5.3 Installing the Control Device

Since it is designed to bypass Plug and Play, the control device cannot be installed or uninstalled using the Plug and Play Manager or DPInst. The control device must be installed and uninstalled using the **wdreg.exe** command-line utility.

To install the control device

1. Disconnect any devices that use DriverCore.
2. Using **wdreg.exe** install the control device INF. For example, if the INF is named **dc_enum_control.inf**, run

```
wdreg.exe -inf dc_enum_control.inf install
```

This creates the control device and sets the **UseControlDevice** registry key [E.4], adding the key if it is not already there. Normally, the control device should be installed before installing the enumerator device.

TIP

Wdreg is provided in two versions: **wdreg.exe** and **wdreg_gui.exe**. Both are run from the command line and provide the same functionality, except that **wdreg_gui.exe** uses dialogue boxes to display messages.

6.5.4 Uninstalling the Control Device

To uninstall the control device

1. Disconnect any devices that use DriverCore.
2. Using **wdreg.exe** uninstall the control device INF. For example, if the INF is named **dc_enum_control.inf**, run

```
wdreg.exe -inf dc_enum_control.inf uninstall
```

This deletes the control device. Normally, the enumerator device should be uninstalled before uninstalling the control device.

Chapter 7

Accessing the ACM/USB2Serial Device

7.1 Overview

Once you have installed the ACM/USB2Serial driver for your device, you can interact with the device from an application. There are two standard types of communication with an ACM/USB2Serial device:

Serial-data transfer: Raw-data transfer. This type of transfer is implemented using the native Windows file-management API [7.2].

Encapsulated-data transfer: Transfer of data that is encapsulated in the proprietary format of the supported CDC interface control protocol [7.3]. You can implement this type of transfer using the DriverCore CDC encapsulated-data transfer functions [I.3.2], provided by the ACM/USB2Serial driver.

7.2 Transferring Serial Data Using the Windows API

DriverCore exposes your ACM/USB2Serial USB device as a COM port (virtual COM port / virtual serial modem). You can, therefore, use the Windows file-management API to open a handle to the device and implement serial-data transfer:

1. Use `CreateFile()` to open the COM port assigned to your USB device, as demonstrated in the following code sample:

```
HANDLE com_h;
...
com_h = CreateFile("\\.\COM19", GENERIC_READ | GENERIC_WRITE, 0,
    NULL, OPEN_EXISTING, 0, NULL);
if (INVALID_HANDLE_VALUE == com_h)
{
    /* Failed to acquire a device handle */
    goto Exit;
}
```

2. Use `ReadFile()` and `WriteFile()` to communicate with the device.
3. Use `CloseHandle()` to close the COM port, as demonstrated in the following code sample:

```
if (!CloseHandle(com_h)) /* com_h: Handle acquired with CreateFile() */
{
    /* Failed to release the device handle */
    goto Exit;
}
```

NOTE

For more information about the Windows file-management API, refer to the Microsoft Developer Network (**MSDN**):

- File Management – [http://msdn.microsoft.com/en-us/library/aa364229\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa364229(VS.85).aspx)
- CreateFile Function – [http://msdn.microsoft.com/en-us/library/aa363858\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa363858(VS.85).aspx)
- Opening a File for Reading or Writing – [http://msdn.microsoft.com/en-us/library/bb540534\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb540534(VS.85).aspx)
- CloseHandle Function – [http://msdn.microsoft.com/en-us/library/ms724211\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms724211(VS.85).aspx)

7.3 Transferring Encapsulated Data Using the DriverCore API

According to the CDC 1.1 Specification, a CDC ACM application can optionally communicate with a device via the transfer of data that is encapsulated in the proprietary format of the supported CDC interface control protocol; the application sends encapsulated commands to the device, and receives encapsulated responses.

The ACM/USB2Serial driver, features the DriverCore CDC API [I.3], which provides specific functions for transferring encapsulated data between the host and the device, as outlined in section I.3.2.

To transfer encapsulated data, the application should

1. Acquire a handle to the device [I.1].
2. Use the DriverCore CDC encapsulated-data transfer functions [I.3.2] to exchange encapsulated data with the device, via the acquired device handle.
3. Release the handle to the device [I.1].

7.3.1 Windows COM Port Access

Windows does not allow COM port sharing; at any given moment, there can only be one open handle to each COM port. Therefore, when implementing both serial and encapsulated-data transfer for your device, you need to either share the same port handle for both types of transfer, or otherwise synchronize the acquiring and releasing of the port handle.

The ACM/USB2Serial driver enables you to support both serial and encapsulated-data transfer, regardless of whether you select to implement both types of transfer from a single application [7.3.3] or from two applications, each dedicated to handling a different type of transfer [7.3.4], as explained in the following sections.

7.3.2 Driver Modes

The ACM/USB2Serial driver supports two driver modes:

- Single-device mode** – In this default driver mode, the ACM/USB2Serial driver creates a single COM port device instance (i.e., virtual COM port or virtual serial modem), which is associated with your physical USB device. This mode is compatible with single-application implementations [7.3.3].

Dual-device mode – In this mode, the ACM/USB2Serial driver creates two device instances for your physical USB ACM/USB2Serial device:

1. A **Serial Device** – A COM port (similar to that created by the driver in the single-device mode) – for transferring serial data
2. An **Extra-Control Device** – for transferring encapsulated data

This mode enables simultaneous access to the same physical USB ACM/USB2Serial device, from two separate applications – each dedicated to handling a different type of data transfer – as explained in section 7.3.4.

To activate the dual-device driver mode, you need to install the Extra-Control Device INF file (default name: **cdc_acm_ser_ctrl.inf**), and set the **ExtraControlEnabled** enumerator driver parameter [E.1] .

7.3.3 Single-Application Implementation

When implementing serial and encapsulated-data transfer in the same application, you can share the device handle (i.e., the handle to the COM port associated with the device) for both types of communication.

In this type of implementation, the application's work flow should be as follows:

1. Acquire a handle to the Serial Device (COM port), as outlined in section I.1.1.
2. Use the acquired handle both to transfer serial data – using the Windows file-management API [7.2] – and to send and receive encapsulated data – using the DriverCore CDC encapsulated-data transfer functions [I.3.2].
3. Release the device handle, as outlined in section I.1.1.

Figure 7.1 illustrates a single-application implementation [7.3.2].

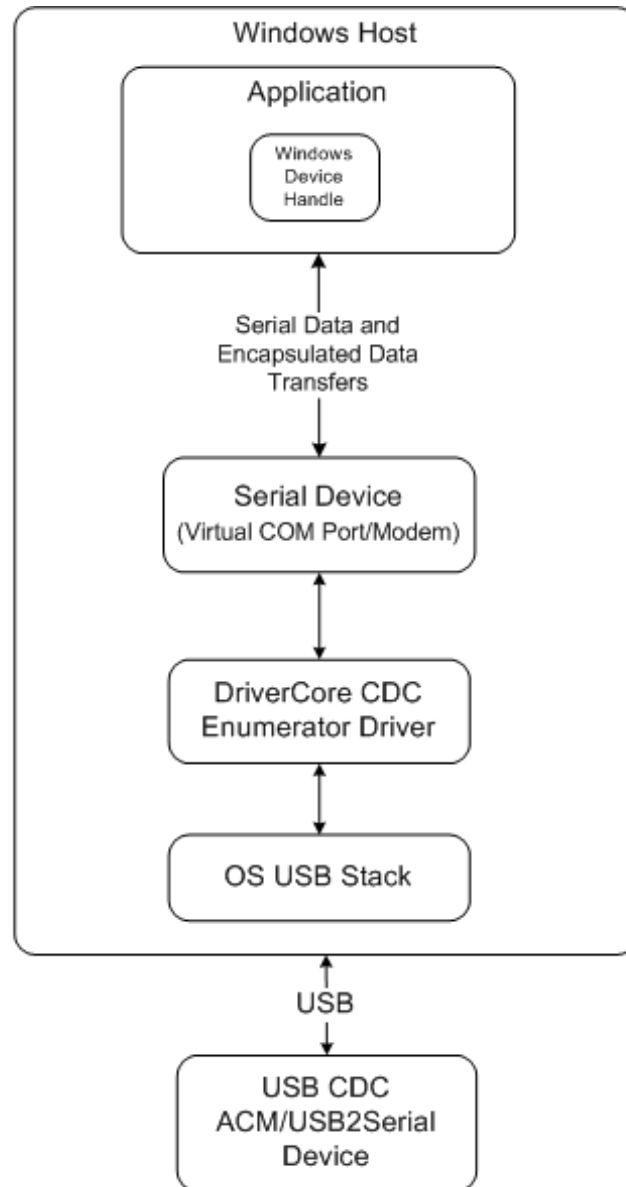


Figure 7.1: Encapsulated-Data Transfer – Single-Application Implementation

7.3.4 Dual-Application Implementation

You may want to separate the handling of the serial- and encapsulated-data transfers into two applications; this is useful, for example, if you wish to use an existing legacy serial application (such as a terminal emulator), and write your own application to transfer encapsulated data. Such a dual-application implementation is not normally possible, due to the Windows restriction on COM-port sharing, as outlined in section 7.3.1. However, the ACM/USB2Serial driver's dual-device driver mode [7.3.2] provides a means for bypassing this problem.

As explained in section 7.3.2, in the dual-device driver mode, the driver creates two device instances – a Serial Device and an Extra-Control Device. Using this driver mode, the physical USB ACM/USB2Serial device can be accessed from two separate applications – each dedicated to handling a different type of data transfer, and communicating with a different device instance:

- The **serial application** communicates with the Serial Device and implements raw serial-data transfer, using the Windows file-management API [7.2].
- The **control application** communicates with the Extra-Control Device using the DriverCore functions [I]; encapsulated-data transfer is implemented using the DriverCore CDC encapsulated-data functions [I.3.2].

NOTE

When using a dual-application implementation, the encapsulated-data transfer must be handled only by the control application; the serial application may not send or receive encapsulated data.

Figure 7.2 illustrates a dual-application implementation, using the ACM/USB2Serial driver's dual-device driver mode [7.3.2].

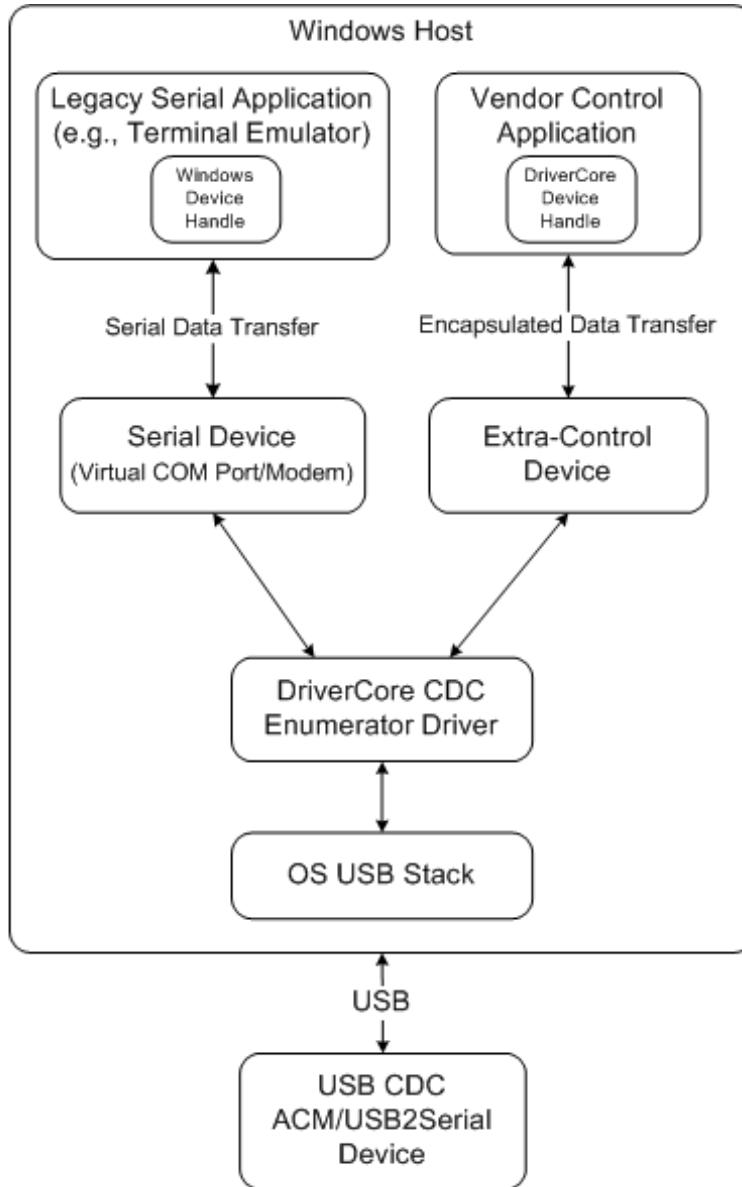


Figure 7.2: Encapsulated-Data Transfer – Dual-Application Implementation

Appendix A

About DriverCore

To promote the USB standard and facilitate the use of USB, the USB Implementers Forum (<http://www.usb.org>) defined USB software protocols (classes) for various device types. The original standard enabled operating systems to communicate with common devices; it defined USB classes such as standard Mass Storage, HID, and Printer. As USB became widespread, additional USB classes were defined, to enable the transport over USB of additional protocols – including Ethernet, Serial, and Bluetooth – providing more flexibility to OEMs and enabling the support of complex multi-function devices.

Jungo's DriverCore class drivers fully implement the USB classes that, in popular operating systems, are either poorly implemented or not supported at all. By helping device manufacturers to expose the functionality of their devices over USB, DriverCore reduces development cost and time-to-market. The ready-to-deploy drivers are fast and robust. Customers repeatedly attest to the value of DriverCore and its ability to accelerate the ramp-up to mass production and shorten time-to-market.

NOTE

The DriverCore drivers are targeted at standard operating systems that provide native support for USB, such as Windows. In addition, for embedded systems that do not inherently support USB, Jungo offers embedded USB host, device, and On-The-Go (OTG) stacks – see the description of the USBware product line: http://www.jungo.com/st/usbware_embedded_usb_solution.html.

The USBware host stack includes embedded CDC class drivers: http://www.jungo.com/st/embedded_usb_cdc.html.

Appendix B

The Customized Driver Package

This chapter describes the contents of the customized DriverCore driver package directory, as generated using the DriverCore wizard [3] or prepared manually [6.1].

NOTE

In the following instructions, replace **<enum>** and **<child>** instances with the names of your selected enumerator driver and child driver, respectively. For example, if the name of your enumerator driver is **jungo_enum**, replace **<enum>.sys** with **jungo_enum.sys**. In addition, replace **<extra_ctrl>** instances with the name of the Extra-Control Device [7.3.2].

The top-level driver package directory (e.g., **x86** or **x64**) contains the following files and directories:

- A **driver** directory
- **DPIInst.exe** – the Microsoft Windows Driver Package Installer (DPIInst) [C]
- **difxapi.dll** – the Microsoft Driver Install Frameworks API (DIFxAPI) DLL, required by DPIInst
- **dpinst.xml** – DPIInst descriptor (configuration) file

The **driver** directory contains the following files:

- **WdfCoInstaller01007.dll** – the Microsoft Windows Driver Foundation (**WDF**) co-installer DLL, which is required for the operation of the DriverCore drivers
- **<enum>.sys** (default: **dc_enum.sys**) – the enumerator driver, which provides the interface between the USB and the child driver
- **<enum>.inf** (default: **dc_enum.inf**) – an INF file for installing the enumerator driver
- **<child>.sys** (default: **cdc_acm.sys**) – the child driver
- **<child>.inf** (default: **cdc_acm_modem.inf** or **cdc_acm_obex.inf**) – an INF file for installing the child driver
- **<enum>.pdb** – a Windows symbols file for debugging the enumerator driver
- **<child>.pdb** – a Windows symbols file for debugging the child driver

The following **driver** directory files are only required if you have selected to use the control device mode – see *Isolating Child Devices from Plug and Play* [6.5]. Otherwise, they need not even be included in your customized driver package:


- **<enum>_control.inf** (default: **dc_enum_control.inf**) – an INF file for creating the control device
- **wdreg.exe** – a utility for installing/uninstalling the control device INF
- **wdreg_gui.exe** – a graphical version of **wdreg.exe**

The following **driver** directory files are only required if you have selected to use the Extra-Control Device feature of the dual-device driver mode [7.3.2]; otherwise, they need not even be included in your customized driver package:

- **<extra_ctrl>.inf** (default: **cdc_acm_ser_ctrl.inf**) – an INF file for creating an Extra-Control Device [7.3.2]

The **driver** directory contains the following subdirectories:

- **enum_sys** – contains files for rebuilding the enumerator driver
- **child_sys** – contains files for rebuilding the child driver

 Appendix D explains how to customize the driver file properties and rebuild the drivers, using the files from the **enum_sys** and **child_sys** directories.

Appendix C

The Windows Driver Package Installer (DPInst)

DriverCore driver packages include the Microsoft Windows Driver Package Installer (**DPInst**), which can be used for the installation and uninstallation of the DriverCore drivers [4]. DPInst has many useful options for streamlining driver installation and uninstallation. Options are configurable by editing the **dpinst.xml** file that is included in the driver package, or via command-line switches.

NOTE

Your customized driver package directory [B] contains the DPInst executable – **DPInst.exe** – and the DPInst configuration file – **dpinst.xml**

For information about the DPInst configuration flags, see <http://msdn.microsoft.com/en-us/library/ms791062.aspx>.

For information about the DPInst command-line switches, see <http://msdn.microsoft.com/en-us/library/ms790806.aspx>.

You can also display the DPInst command-line switches using the **/help** switch:
DPInst.exe /help

For more information about DPInst, see <http://msdn.microsoft.com/en-us/library/ms790308.aspx>.

Appendix D

Customizing Driver File Properties via the Resource Files

The enumerator and child driver (**.sys**) files each contain properties (strings) that can be modified by editing the driver's resource (**.rc**) file and rebuilding the driver. Your customized driver package [B] contains the resource files for your enumerator and child drivers, and the other files required for rebuilding these drivers using the Windows Driver Kit (**WDK**).

To edit a driver's resource file and rebuild the driver using the WDK

1. Navigate to the **driver** subdirectory in your driver package directory.
2. Navigate to the subdirectory containing the resource file for the target driver: **enum_sys** for the enumerator driver, or **child_sys** for the child driver.
3. Edit the resource (**.rc**) file to customize the property definitions.
4. Rebuild the driver. The **enum_sys** and **child_sys** directories contain the files required for building the driver using the Windows Driver Kit (WDK), available from Microsoft.

To rebuild the driver using the WDK

- (a) Open a command prompt to the directory containing the driver to be built (**enum_sys** for the enumerator driver; **child_sys** for the child driver).
- (b) At the command prompt, type **build** to build the driver.
The driver is created in a new subdirectory.

Appendix E

Manually Customizing the Drivers via the INF Files

Windows drivers are installed via INF files. The INF file includes add-registry sections, which contain entries for modifying the Windows registry keys. The INF **AddReg** directives determine which add-registry sections to process, thus controlling the driver parameters that will be defined in the registry when the INF file is installed.

You can customize the DriverCore drivers, before their installation, by defining your desired driver parameters in the relevant add-registry INF sections.

Some configurable parameters may already be defined in the INF file; you can edit the existing definitions, as well as add new parameter definitions.

To define a registry parameter in the INF file, under the relevant **AddReg** directive, add a line that conforms to the following format:

```
HKR, Parameters, <param_name>, <param_type>, <param_value>
```

(see the MSDN documentation for additional information:

<http://msdn.microsoft.com/en-us/library/ms794514.aspx>).

For example, the template and wizard-generated DriverCore INF files include the following **DebugLevel** parameter definition, under **[DevInstall.AddService.AddReg]**:

```
HKR, Parameters, DebugLevel, 0x00010001, 2
```

NOTE

Unless otherwise specified, the type of all registry parameters documented in this chapter is **REG_DWORD**; this type is indicated by setting the **0x00010001** flag as the **<param_type>** in the registry parameter definition, as demonstrated above for the **DebugLevel** parameter.

The following sections describe configurable driver parameters that you can define by editing the DriverCore driver INF files.

The documented default parameter values are the values that will be used when the INF file does not define the related parameter; not all parameters have default values.

NOTE

Most of the configurable driver parameters can also be customized via the DriverCore wizard, when using the wizard to generate the driver package [3].

E.1 Custom DriverCore Driver Parameters

This section describes custom DriverCore driver parameters. These parameters can be defined in the [DevInstall.AddService.AddReg] section of the relevant INF file.

The following parameters can be defined independently for each DriverCore driver (e.g., enumerator and child), by editing the driver's INF file:

Parameter	Description	Default Value
DebugLevel	Debug level: controls the types of debug information that the DriverCore driver sends to the operating system [J.5]	2

The following parameters can be defined in the enumerator INF file:

Parameter	Description	Default Value
IdleTimeout	Number of idle seconds before the device enters suspend mode (0 = suspend disabled). Note: For details and limitations, see <i>Power Management</i> [6.3].	20
ExtraControlEnabled	Enable the dual-device driver mode [7.3.2]	1

E.2 USB Hardware Parameters

The USB hardware parameters can be defined in the [DevInstall_NT_HW_AddReg] section of the enumerator INF file.

Parameter	Description	Default Value
EnableZLP	Send a zero-length packet after each OUT transfer whose byte count is a multiple of the maximum packet size	1

E.3 Device Hardware Parameters

The hardware parameters described in this section can be defined in the [DevInstall_NT_HW_AddReg] section of the relevant INF file.

NOTE

Unless otherwise specified, the parameters described in this section apply to the CDC ACM child device, and do *not* apply to the Extra-Control Device of the dual-device driver mode [7.3.2].

E.3.1 General Parameters

The following device hardware parameters can be defined in the enumerator INF file:

Parameter	Description	Default Value
DeviceClassType	<p>Forcibly sets the device class type of the child device, ignoring any type definition in the device descriptors. The following device-class-type values are valid:</p> <ul style="list-style-type: none"> 0 – UNDEFINED 1 – ACM 2 – ECM 3 – OBEX 4 – EEM 5 – ICCD 6 – DFU <p>Note: Setting this parameter to a specific device class type does not guarantee standard class behavior if the device is not implemented in accordance with the specification of the selected class type.</p>	0

The following device hardware parameters can be defined in the child INF file:

Parameter	Description
DeviceGUID	<p>A custom globally unique identifier (GUID) to assign to the device, in addition to the GUID assigned by default.</p> <p>When defining this parameter, its type must be set to %REG_SZ%, and its value must be indicated within curly braces – {<GUID>}; i.e., define it as follows: HKR, , "DeviceGUID", %REG_SZ%, {<GUID>} (where "<GUID>" is your desired GUID).</p> <p>Note: This parameter can be set for any child device, including the Extra-Control Device of the dual-device driver mode [7.3.2]. By default, DriverCore assigns a GUID of {cd56b1bd-c817-4e15-88de-bc4af1202dcb} to the Extra-Control Device.</p>

E.3.2 CDC ACM Child-Device Parameters

The following CDC ACM child-device hardware parameters can be defined in the child INF file:

Parameter	Description	Default Value
DataInPacketsPerXfer	The number of USB packets per data IN transfer	8
DataInBuffers	The number of transfers that the data IN endpoint can buffer	0x80
CtrlDevHoldChildIfInUse	Delays unloading the CDC ACM child device upon unloading of the enumerator device, until there are no open handles to the child device. This feature requires control device mode – see <i>Isolating Child Devices from Plug and Play</i> [6.5].	0
CallClearData	Resets the IN and OUT data pipes (but not the notification pipe), when any of the following events occurs: <ul style="list-style-type: none"> • A handle is opened to the device. • The device wakes up (resumes) from suspend, and there is an open handle to the device. 	0

E.3.3 Non-Standard-CDC-Device Parameters

Some of the CDC class drivers define parameters that are used for non-standard CDC devices. The driver applies these parameters when it detects a configuration descriptor that does not comply with the CDC 1.1 Specification; for standard CDC devices, these parameters are ignored. When implementing non-standard CDC devices, the desired configuration must be defined by modifying the parameter values.

The following non-standard-CDC-device hardware parameters can be defined in the enumerator INF file:

Parameter	Description
Configuration	USB configuration to use
DataInterface	Data interface
DataAlternateEnable	Alternate setting that enables the data interface *
DataAlternateDisable	Alternate setting that disables the data interface *
DataInEp	The address of the data IN endpoint's "enable" alternate setting (DataAlternateEnable)
DataOutEp	The address of the data OUT endpoint's "enable" alternate setting (DataAlternateEnable)
CommInterface	Communication interface (optional)
NotifyEp	The address of the notification endpoint

* Note: Devices that do not support enable/disable should specify identical values for **DataAlternateEnable** and **DataAlternateDisable**.

E.4 Control Device Hardware Parameters

The following hardware parameters can be defined in the [`<enum>.AddReg`] section of the control device INF file.

NOTE

Replace `<enum>` with the name of your enumerator driver. For example, if its name is `jungo_enum`, the section name would be `[jungo_enum.AddReg]`.

Parameter	Description	Default Value
UseControlDevice	Enables control device mode	1

Appendix F

Configuring Device Settings using the Windows Device Manager

The device settings of some devices can be modified via the Device Manager.

For example, to modify the port settings of a virtual COM port

1. Open the Device Manager (**devmgmt.msc**).
2. Right-click the port to configure, then select *Properties*.
The Port Properties window opens.
3. Select the *Port Settings* tab.
4. Configure the desired parameters, then click *OK*.

Appendix G

Providing CDC ACM Functionality on Windows 2000

According to the CDC 1.1 Specification, an ACM device should be implemented using two interfaces that provide a single function. This is typically implemented by uniting the interfaces using an Interface Association Descriptor (IAD) (see the MSDN IAD documentation: <http://msdn.microsoft.com/en-us/library/aa475772.aspx>).

However, the Windows 2000 composite device driver – which is responsible for handling multi-interface devices – does not support IADs. As a result, devices that contain IADs are not enumerated correctly on this OS, and it is impossible to successfully install drivers for them.

To bypass this problem, the DriverCore ACM/USB2Serial driver also supports an alternative device implementation for simulating standard CDC ACM device behavior:

A single interface, with one alternate setting that contains the following pipes:

- 1 interrupt pipe – for communication control
- 2 bulk pipes – for communication data

Appendix H

Automatic Repair of Modem-Device Symbolic Links

There is a bug in the Windows **modem.sys** driver. When a USB modem device is disconnected and then reconnected, while the COM port is held open, **modem.sys** fails to update the device's symbolic link. This can result in dial attempts failing with error 633.

To eliminate this issue, the DriverCore ACM/USB2Serial driver automatically fixes the symbolic link. This behavior is transparent to the user, and dialing can proceed without errors.

Appendix I

API Reference

This appendix describes DriverCore APIs provided by the ACM/USB2Serial driver.

I.1 Acquiring and Releasing Device Handles

To communicate with your device from an application, you must first acquire a handle to the device.

When working in the dual-device driver mode [7.3.2], you need to acquire handles both to the Serial Device and to the Extra-Control Device.

When a device handle is no longer required, it should be released using the appropriate function.

The following sections explain how to acquire and release device handles when working with the ACM/USB2Serial driver.

I.1.1 Serial-Device Handle

To acquire a handle to the Serial Device – either from the serial application, in a dual-application implementation [7.3.4], or when using a single-application implementation [7.3.3] – use one of the following methods:

- To identify the device by its COM port name, call the Windows **CreateFile()** function [7.2].
- To identify the device by its GUID, use the DriverCore common-API device-handle-acquisition functions, as outlined in section I.2.1.

The acquired device handle can be passed both to the native Windows file-management functions [7.2] – for serial-data transfer – and to the DriverCore CDC encapsulated-data transfer functions [I.3.2] – for encapsulated-data transfer (when using a single-application implementation [7.3.3]).

When the handle is no longer required, release it using the method that matches the handle-acquisition method that you used:

- If you used the Windows `CreateFile()` function to acquire the device handle, release the handle by calling the Windows `CloseHandle()` function [7.2].
- If you used the DriverCore common-API functions to acquire device handles, release these handles by calling the the matching DriverCore common-API handle-release functions, as outlined in section I.2.1.

I.1.2 Extra-Control Device Handle

To acquire a handle to the Extra-Control Device – when working in the dual-device driver mode [7.3.2] – use one of the following alternatives:

- To identify the Extra-Control Device by the name of the COM port (Serial Device) with which it is associated, call the DriverCore CDC API `dc_acm_ctrl_handle_get()` function [I.3.1.1], as explained in section I.3.1.
- To identify the Extra-Control Device by its GUID, use the DriverCore common-API device-handle-acquisition functions, as outlined in section I.2.1.

The acquired device handle can be passed to the DriverCore CDC encapsulated-data transfer functions [I.3.2].

When the handle to the Extra-Control Device is no longer required, release it using the method that matches the handle-acquisition method that you used:

- If you acquired the handle using the DriverCore CDC API `dc_acm_ctrl_handle_get()` function, release the handle by calling the DriverCore CDC API `dc_acm_ctrl_handle_put()` function [I.3.1.2], as explained in section I.3.1.
- If you used the DriverCore common-API functions to acquire device handles, release these handles by calling the the matching DriverCore common-API handle-release functions, as outlined in section I.2.1.

I.2 DriverCore Common API

This section describes the DriverCore common API, which is declared in the `api_common.h` header file.

I.2.1 Acquiring and Releasing Device Handles

This section describes the DriverCore common-API functions for acquiring and releasing device handles.

NOTE

The functions described in this section allow you to acquire a device handle by specifying the device's GUID. Alternatively, as explained in section I.1, you can use the native Windows API (for the Serial Device) or the DriverCore CDC API (for the Extra-Control Device [7.3.2]) to acquire a device handle by specifying the related COM port name.

To acquire a device handle using the DriverCore common API, perform the following sequence:

1. Call `dc_devices_get()` [I.2.1.1] to acquire a list of handles to internal DriverCore device structures ("DriverCore device handles"), which match the target device's GUID.

NOTE

You can use the `DeviceGUID` parameter, in the child INF file, to set a custom GUID for the target device, as explained in section E.3.1.

2. Identify the handle to your target device, from among the list of handles returned by `dc_devices_get()`, and pass this handle to `dc_device_open()` [I.2.1.3] to receive a Windows handle to the device.

The following code sample demonstrates how to use the DriverCore common-API functions to acquire a Windows handle to the Extra-Control Device.

NOTE

The GUID in the sample is set using the `GUID_ACM_SERIAL_CTRL_IFC` preprocessor definition, which is assigned the default GUID of the Extra-Control Device – `0xcd56b1bd, 0xc817, 0x4e15, 0x88, 0xde, 0xbc, 0x4a, 0xf1, 0x20, 0x2d, 0xcd`.

```

static const GUID guid = GUID_ACM_SERIAL_CTRL_IFC;
device_h *dev_list = NULL;
HANDLE dev_h = INVALID_HANDLE_VALUE;
int dev_count = 0, status;
...
/* Acquire a list of devices, by GUID */
status = dc_devices_get(guid, &dev_list, &dev_count);
if (DC_STATUS_SUCCESS != status)
{
    /* Failed to acquire a list of matching devices */
    goto Exit;
}

if (!dev_count)
    /* Failed to locate a matching device */
    status = DC_STATUS_DEVICE_NOT_CONNECTED;
    goto Exit;
}

/* Identify the device handle that matches your target device;
the following code assumes that this is the first handle
in the returned *dev_handles list. */

/* Acquire a Windows device handle to your target device */
dev_h = dc_device_open(dev_handles[0], 0);
if (INVALID_HANDLE_VALUE == dev_h)
{
    /* Failed to open a communication channel with the device */
    goto Exit;
}

```

The acquired Windows device handle can be passed to the native Windows file-management functions [7.2] – in the case of a Serial-Device handle – and to the DriverCore CDC encapsulated-data transfer functions [I.3.2] – in the case of an Extra-Control Device handle in the dual-device driver mode [7.3.2], or a Serial-Device handle in the single-device driver mode [7.3.2].

When the acquired handles are no longer required, release them in the following manner:

1. Release the Windows device handle returned by `dc_device_open()`, by passing it to `dc_device_close()` [I.2.1.4].
2. Release the list of DriverCore device handles returned by `dc_devices_get()`, by passing it to `dc_devices_put()` [I.2.1.2].

The following code sample demonstrates how to release the handles acquired using the DriverCore common-API functions:

```
if (INVALID_HANDLE_VALUE != dev_h)
{
    /* Release the Windows device handle acquired using dc_device_open() */
    status = dc_device_close(dev_h);

    if (DC_STATUS_OPERATION_FAILED == status)
    {
        /* Failed to close the communication with the device */
    }
}

if (dev_count)
{
    /* Release the list of DriverCore device handles acquired using
       dc_devices_get(). */
    status = dc_devices_put(&dev_handles, dev_count);

    if (DC_STATUS_SUCCESS == status)
    {
        /* Failed to release the DriverCore device handles */
    }
}
```

I.2.1.1 dc_devices_get()

PURPOSE

Identifies devices with the specified GUID, and returns a list of handles to internal DriverCore device structures ("DriverCore device handles") for the matching devices.

PROTOTYPE

```
int WINAPI dc_devices_get(
    GUID guid,
    device_h **list,
    int *count);
```

PARAMETERS

Name	Type	Input/Output
➤ guid	GUID	Input
➤ list	device_h**	Output
➤ count	int*	Output

DESCRIPTION

Name	Description
➤ guid	A GUID structure depicting the globally unique identifier (GUID) for which to search (see the MSDN GUID structure documentation: http://msdn.microsoft.com/en-us/library/aa373931%28VS.85%29.aspx). Note: The GUID for the Extra-Control Device must be either a custom GUID that was set for this device via the DeviceGUID registry key [E.3.1], or the default Extra-Control Device GUID – 0xcd56b1bd, 0xc817, 0x4e15, 0x88, 0xde, 0xbc, 0x4a, 0xf1, 0x20, 0x2d, 0xcd – which is assigned to the GUID_ACM_SERIAL_CTRL_IFC preprocessor definition.
➤ list	A pointer to the address of the list of matching DriverCore device handles, created by the function. Note: If no matching devices are found (*count=0), the function does not update this parameter.
➤ count	A pointer to the number of matching devices found by the function.

RETURN VALUE

Returns DC_STATUS_SUCCESS (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.2.1.2 dc_devices_put()**PURPOSE**

Releases a list of DriverCore device handles, created by a previous call to `dc_devices_get()` [I.2.1.1].

PROTOTYPE

```
int WINAPI dc_devices_put(
    device_h **list,
    int count);
```

PARAMETERS

Name	Type	Input/Output
➤ list	device_h**	Output
➤ count	int	Input

DESCRIPTION

Name	Description
➤ list	A pointer to the address of the DriverCore device-handles list to release.
➤ count	The number of handles in the returned list .

RETURN VALUE

Returns `DC_STATUS_SUCCESS` (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.2.1.3 dc_device_open()**PURPOSE**

Opens a communication channel with the specified device.

PROTOTYPE

```
HANDLE WINAPI dc_device_open (
    device_h dev,
    DWORD options);
```

PARAMETERS

Name	Type	Input/Output
> dev	device_h	Input
> options	DWORD	Input

DESCRIPTION

Name	Description
> dev	DriverCore device handle, as received from a previous call to <code>dc_devices_get()</code> [I.2.1.1]
> options	Reserved for future use

RETURN VALUE

Returns a Windows device handle on success, or `INVALID_HANDLE_VALUE` on failure

I.2.1.4 dc_device_close()**PURPOSE**

Closes the communication channel with the specified device.

PROTOTYPE

```
int WINAPI dc_device_close (HANDLE dev_h);
```

PARAMETERS

Name	Type	Input/Output
> dev_h	HANDLE	Input

DESCRIPTION

Name	Description
> dev_h	Windows device handle [I.1]

RETURN VALUE

Returns DC_STATUS_SUCCESS (0) on success, or DC_STATUS_OPERATION_FAILED on failure

I.3 DriverCore CDC API

This section describes the DriverCore CDC API, which is declared in the `cdc_api.h` header file.

I.3.1 Acquiring and Releasing Extra-Control Device Handles

This section explains how to use the DriverCore CDC API to acquire and release a handle to the Extra-Control Device, when working in the dual-device driver mode [7.3.2]. This handle can then be passed to the DriverCore CDC encapsulated-data-transfer functions [I.3.2].

NOTE

As explained in section I.1, you can alternatively use the DriverCore common-API functions to acquire and release a handle to the Extra-Control Device, identifying the device by its GUID instead of by the name of the COM port with which it is associated.

To acquire a handle to the Extra-Control Device using the DriverCore CDC API, call the `dc_acm_ctrl_handle_get()` function [I.3.1.1], as demonstrated in the following code sample:

```
HANDLE extra_ctrl_h;
...
/* Acquire a handle to the Extra-Control Device */
extra_ctrl_h = dc_acm_ctrl_handle_get(L"COM19");
if (INVALID_HANDLE_VALUE == extra_ctrl_h)
{
    /* Failed to acquire a handle to the Extra-Control Device;
    perform error handling here */
    ...
}
```

When the handle is no longer required, release it by calling the DriverCore CDC API `dc_acm_ctrl_handle_put()` function [I.3.1.2], as demonstrated in the following code sample:

```
/* Release the Extra-Control Device handle acquired using
dc_acm_ctrl_handle_get() */
if (DC_STATUS_SUCCESS != dc_acm_ctrl_handle_put(extra_ctrl_h))
{
    /* Failed to release the Extra-Control Device handle;
    perform error handling here */
    ...
}
```

I.3.1.1 dc_acm_ctrl_handle_get()**PURPOSE**

Acquires a Windows handle to an Extra-Control Device.

NOTE

This function should only be used to acquire a handle to the Extra-Control Device, when working in the dual-device driver mode [7.3.2]; it should not be used to acquire a handle to the Serial Device.

When the application no longer needs to access the Extra-Control Device, it should release the handle using the `dc_acm_ctrl_handle_put()` function [I.3.1.2].

i The function's prototype differs depending on whether the application is built with Unicode support or not, as shown below.

PROTOTYPE (UNICODE)

```
HANDLE WINAPI dc_acm_ctrl_handle_get(WCHAR *com_port);
```

PARAMETERS (UNICODE)

Name	Type	Input/Output
> com_port	WCHAR *	Input

PROTOTYPE (NON-UNICODE)

```
HANDLE WINAPI dc_acm_ctrl_handle_get(CHAR *com_port);
```

PARAMETERS (NON-UNICODE)

Name	Type	Input/Output
> com_port	CHAR *	Input

DESCRIPTION

Name	Description
> com_port	The name of the COM port (Serial Device) with which to associate the Extra-Control Device (e.g., <code>L"COM19"</code>).

RETURN VALUE

Returns a Windows device handle on success, or `INVALID_HANDLE_VALUE` on failure

I.3.1.2 dc_acm_ctrl_handle_put()**PURPOSE**

Releases a Windows handle to an Extra-Control Device.

NOTE

This function should only be used to release a handle to an Extra-Control Device, acquired using `dc_acm_ctrl_handle_get()` [I.3.1.1], when working in the dual-device driver mode [7.3.2].

PROTOTYPE

```
int WINAPI dc_acm_ctrl_handle_put(HANDLE dev_h);
```

PARAMETERS

Name	Type	Input/Output
> dev_h	HANDLE	Input

DESCRIPTION

Name	Description
> dev_h	Windows device handle [I.1]

RETURN VALUE

Returns `DC_STATUS_SUCCESS` (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.3.2 Encapsulated-Data Transfer Functions

Use the DriverCore CDC encapsulated-data transfer functions, described in this section, to send and receive encapsulated data:

- Use `dc_cdc_encapsulated_send()` [I.3.2.1] to send encapsulated data (a command) to the device.
- Use `dc_cdc_encapsulated_wait()` [I.3.2.3] to wait until an encapsulated response is available from the device.
- Use `dc_cdc_encapsulated_get()` [I.3.2.2] to receive encapsulated data (a response) from the device.

The following code sample demonstrates an encapsulated-data transfer using the DriverCore CDC encapsulated-data transfer functions.

```
/* Send an encapsulated command to the device */
if (DC_STATUS_SUCCESS != dc_cdc_encapsulated_send(
    dev_h, my_cmd_buf, my_cmd_buf_length))
{
    /* Failed to send the command; perform error handling here */
    ...
}

/* Wait for a response from the device */
if (DC_STATUS_SUCCESS != dc_cdc_encapsulated_wait(dev_h))
{
    /* Failed to wait for the response; perform error handling here */
    ...
}

/* Receive the encapsulated response from the device */
else if (DC_STATUS_SUCCESS != dc_cdc_encapsulated_get(
    dev_h, my_response_buf_response_buf_length, response_length))
{
    /* Failed to receive the response; perform error handling here */
}
```

I.3.2.1 dc_cdc_encapsulated_send()**PURPOSE**

Sends a SendEncapsulatedCommand request, in order to issue an encapsulated command to the device.

PROTOTYPE

```
int WINAPI dc_cdc_encapsulated_send(
    HANDLE dev_h,
    PVOID cmd_buf,
    DWORD cmd_buf_len);
```

PARAMETERS

Name	Type	Input/Output
➤ dev_h	HANDLE	Input
➤ cmd_buf	PVOID	Input
➤ cmd_buf_len	DWORD	Input

DESCRIPTION

Name	Description
➤ dev_h	Windows device handle [I.1]
➤ cmd_buf	Pointer to a buffer containing the encapsulated command to send
➤ cmd_buf_len	Size, in bytes, of the encapsulated command

RETURN VALUE

Returns DC_STATUS_SUCCESS (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.3.2.2 dc_cdc_encapsulated_get()

Sends a GetEncapsulatedResponse request, in order to receive an encapsulated response from the device.

PROTOTYPE

```
int WINAPI dc_cdc_encapsulated_get(
    HANDLE dev_h,
    PVOID cmd_response_buf,
    DWORD cmd_response_buf_len,
    PDWORD bytes_transferred);
```

PARAMETERS

Name	Type	Input/Output
➤ dev_h	HANDLE	Input
➤ cmd_response_buf	PVOID	Output
➤ cmd_response_buf_len	DWORD	Input
➤ bytes_transferred	PDWORD	Output

DESCRIPTION

Name	Description
➤ dev_h	Windows device handle [I.1]
➤ cmd_response_buf	Pointer to a buffer to be filled with the received response
➤ cmd_response_buf_len	Size, in bytes, of the response buffer (cmd_response_buf)
➤ bytes_transferred	Pointer the response size, in bytes

RETURN VALUE

Returns DC_STATUS_SUCCESS (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.3.2.3 dc_cdc_encapsulated_wait()**PURPOSE**

Waits for a ResponseAvailable notification from the device, indicating that a GetEncapsulatedResponse request should be issued [I.3.2.2].

 The function blocks until a response is available from the device.

PROTOTYPE

```
int WINAPI dc_cdc_encapsulated_wait (HANDLE dev_h);
```

PARAMETERS

Name	Type	Input/Output
> dev_h	HANDLE	Input

DESCRIPTION

Name	Description
> dev_h	Windows device handle [I.1]

RETURN VALUE

Returns DC_STATUS_SUCCESS (0) on success, or a non-zero DriverCore status code [I.4.2] on failure

I.4 DriverCore Status Codes API

The API described in this section is defined in the `dc_status.h` header file.

I.4.1 `dc_status2str()`

PURPOSE

Retrieves the description of a given DriverCore status code [I.4.2].

PROTOTYPE

```
static const char *dc_status2str(dc_status_t dc_status);
```

PARAMETERS

Name	Type	Input/Output
> <code>dc_status</code>	<code>dc_status_t</code>	Input

DESCRIPTION

Name	Description
> <code>dc_status</code>	DriverCore status code [I.4.2]

RETURN VALUE

Returns a null-terminated string that describes the given status code (`dc_status`)

I.4.2 `dc_status_t` – DriverCore Status Codes

This section describes the DriverCore status codes.

```
typedef int dc_status_t;
```

Name	Description
DC_STATUS_SUCCESS	Success (0)
DC_STATUS_OPERATION_FAILED	Operation failed
DC_STATUS_TIMEOUT	Timeout occurred
DC_STATUS_BUFFER_TOO_SMALL	Buffer too small
DC_STATUS_INVALID_PARAMETER	Invalid parameter
DC_STATUS_INVALID_DEVICE_REQUEST	Invalid device request
DC_STATUS_INVALID_DEVICE_HANDLE	Invalid device handle
DC_STATUS_ACCESS_VIOLATION	Access violation
DC_STATUS_DEVICE_BUSY	Device busy
DC_STATUS_DEVICE_NOT_CONNECTED	Device not connected

TIP

Use `dc_status2str()` [I.4.1] to retrieve the description of a given DriverCore status code.

Appendix J

Troubleshooting

J.1 Troubleshooting FAQ

J.1.1 My drivers fail to install

Verify the hardware IDs defined by the device drivers [J.2].

J.1.2 My composite USB device is not working properly

Follow the instructions in the *Troubleshooting a Composite USB Device* section [J.3].

J.1.3 Windows does not recognize the multiple functions of my composite USB device

Follow the instructions in the *Troubleshooting a Composite USB Device* section [J.3].

J.1.4 My USB device fails to respond to read and write requests

See the *Troubleshooting COM Port Problems* section [J.4].

J.1.5 The Windows Device Manager shows an error ("!") for the COM port/modem associated with my device

See the *Troubleshooting COM Port Problems* section [J.4].

J.1.6 The Windows Device Manager indicates a port conflict for the COM port associated with my device

See the *Troubleshooting COM Port Problems* section [J.4].

J.1.7 How can I get additional help?

See the *Getting Help* section [J.6].

J.2 Verifying the Hardware IDs

If the hardware IDs defined by the drivers for a USB device are incorrect, the operating system will not be able to load the correct drivers for the device. You should therefore verify that the hardware IDs defined for your device in the driver INF files, match the actual device hardware IDs.

To verify the hardware IDs defined by the device drivers

1. View the actual hardware IDs (as reported by the device) in the Device Manager:
 - (a) Open the Windows Device Manager (**devmgmt.msc**), right-click your device, and select **Properties** to display the device Properties window.
 - (b) Select the **Details** tab and choose the **Hardware Ids** option in the drop-down box, to display the device's hardware IDs.
2. Verify that the hardware IDs in the enumerator and child INF files for your device match the actual hardware IDs, as displayed in the Device Manager.

The hardware IDs are defined in the following sections of the INF files (see section 6.1, step #10c):

- [DeviceList]
- [DeviceList.NTx86]
- [DeviceList.NTamd64]

NOTE

- Pay special attention to the Vendor ID (VID) and Product ID (PID).
- If the hardware is a composite device, verify that hardware IDs are properly defined for each device function.

3. If the hardware IDs in the INF files are incorrect, either generate a new driver package using the DriverCore wizard [3], or manually edit the hardware IDs in the INF files (see section 6.1, step #10c).

J.3 Troubleshooting a Composite USB Device

When a composite USB device is connected to a Windows host, the OS normally identifies it as a composite device and installs the Windows USB composite class driver (**usbccgp.sys**) for the device. This driver enumerates each of the device's USB functions; as each function is exposed, a driver can be installed for that function.

If you encounter problems using a composite USB device with your DriverCore drivers, this is probably due to one of the following reasons:

- There is a mistake in the device descriptors, which either prevents the OS from recognizing the device as a composite device and installing the Windows USB composite class driver, or causes the device functions to be enumerated incorrectly [J.3.3].
- The Windows USB composite class driver has been uninstalled (perhaps inadvertently).
- Another driver has been installed for the root device, instead of the Windows USB composite class driver. This can happen, for example, if you have manually edited the INF files in your driver package, and set the hardware IDs in one or more of these files in a manner that applies to the root device instead of to specific device functions. For more information on setting the hardware IDs in the INF files, refer to section 6.1, step #10c.

To troubleshoot problems with a composite USB device

1. Check the device enumeration [J.3.1].
2. If the device's parent driver is not **usbccgp.sys**, install this driver [J.3.2].
3. Recheck the device enumeration [J.3.1].
4. If the device's parent driver is still not **usbccgp.sys**, check the device enumeration [J.3.1] on a different machine, on which the DriverCore drivers have not been installed. If the enumeration on the other machine is successful, you should try to debug the installation on the machine on which you encountered the problems; you can contact Jungo for assistance [J.6].
5. If the parent driver is also incorrect on the other machine, or if the device functions are not displayed or they're displayed incorrectly – verify the device descriptors [J.3.3].
6. If the **usbccgp.sys** driver is installed, and the device-functions information is displayed correctly, verify the hardware ID entries in your DriverCore INF files (see information in section 6.1, step #10c), and reinstall the DriverCore drivers for your device [4.1]. Note: Make sure that the hardware IDs in the INF file are set for specific device functions, and not for the root device.

J.3.1 Checking the Enumeration of a Composite Device

When the drivers for a composite USB device are properly installed, the Windows Device Manager shows the Windows USB composite class driver (**usbccgp.sys**) installed as the device's parent driver, and the device subtree shows the exposed device functions; for each function that has an installed driver, the subtree lists the function description that was assigned, via the INF file, during the driver installation.

To check the enumeration of a composite USB device

1. Connect the device to the computer.
2. Open the Windows Device Manager (**devmgmt.msc**).
3. From the **View** menu, select **Devices by connection**.
4. Locate and right-click the device in the device tree, and select **Properties**.
The device Properties window opens.
5. Select the **Driver** tab.
6. Click **Driver Details...**
The Driver File Details window opens.
7. Verify that the **Driver files** list includes **usbccgp.sys**. If it does not, install this driver by following the instructions in section J.3.2, and then recheck the device enumeration.
8. Close the Driver File Details and device Properties windows, and expand the device subtree.
9. Verify that the device's USB functions are displayed correctly in the device subtree. If they are not, verify the device descriptors [J.3.3].

J.3.2 Installing the USB Composite Class Driver

If the Windows USB composite class driver (**usbccgp.sys**) is not installed as the parent driver of your composite USB device (i.e., for the root device) [J.3.1], uninstall the existing device drivers, and let Windows install **usbccgp.sys**:

1. Connect the device to the host machine.
2. Open the Windows Device Manager (**devmgmt.msc**).
3. Locate and right-click the device in the device tree, and select **Uninstall**.
4. From the **Action** menu, select **Scan for hardware changes**.
The Windows Plug and Play Manager detects new hardware and opens the Found New Hardware Wizard.
5. Follow the instructions of the wizard, and direct the wizard to automatically locate and install the driver.
6. Verify the installation by checking the list of drivers installed for the device [J.3.1].

J.3.3 Verifying the Device Descriptors of a Composite Device

If the operating system is unable to recognize the device as composite and install the correct driver for it, or if the driver is installed but the device functions are not enumerated correctly, you should verify that the configuration of the device descriptors

- Complies with the USB Specification (*Universal Serial Bus Specification, Revision 2.0, April 27th, 2000* – http://www.usb.org/developers/docs/usb_20_05122006.zip)
- Is supported by the operating system

NOTE

The Windows 2000 version of **usbccgp.sys** does not support interface association descriptors (IADs); on Windows 2000, devices that contain IADs are not enumerated correctly, and it is impossible to successfully install drivers for them. For more information about this limitation and how to bypass it, refer to appendix G.

To verify the device descriptors of a composite USB device

1. Install a software or hardware USB analyzer, and configure it to capture the USB traffic that your USB device will generate.

2. Connect your composite USB device to the host machine.
The host enumerates the device, and the device sends its descriptors to the host.
3. In the USB analyzer, turn off USB traffic capture and save the captured traffic to a file.
4. In the capture file, identify the descriptors sent by your device during its enumeration.
5. Verify that the device descriptors properly define the device as a composite device.
6. Verify that the device descriptors properly define the USB functions of the composite device.
7. Verify that the device descriptors are supported in the host's operating system (see the Microsoft USB FAQs – http://www.microsoft.com/whdc/connect/usb/usbfaq_intermed.mspx – and the note above regarding IAD support on Windows 2000).

NOTE

If you need assistance, email the analyzer capture file to Jungo for analysis [K].

J.4 Troubleshooting COM Port Problems

To troubleshoot problems with the COM port associated with your device

1. Check for a COM port conflict [J.4.2].
2. If there is a conflict, assign a different COM port to the device [J.4.3].

NOTE

A modem is a type of COM port; therefore, **COM port** references in this section also apply to modems.

J.4.1 COM Port Conflicts – Overview

Windows maintains a COM port database (**ComDB**) for arbitrating the use of COM port numbers in the system, and ensuring that the same COM port number is not assigned twice. This database is used by all Windows-supplied installers, and Microsoft recommends that vendor-supplied installers also use it (see the MSDN documentation: <http://msdn.microsoft.com/en-us/library/ms800603.aspx>).

The DriverCore drivers use the Windows COM port database to assign COM ports numbers, as recommended by Microsoft. However, some drivers (mainly old drivers)

do not follow Microsoft's recommendation, and reserve port numbers for their devices independently, without updating the OS COM port database. As a result, the same number can end up being assigned to multiple ports, causing a COM port conflict:

- One of the COM ports will be inaccessible.
- The device on the inaccessible port will fail to respond to read/write requests.
- The Device Manager will usually display an error ("!") for the inaccessible port.
- The *Device status*, in the *General* tab of the Device Manager's Port Properties window, will indicate a conflict.

To resolve a port number conflict, assign a different COM port to your device [J.4.3].

J.4.2 Checking for a COM Port Conflict

As explained above [J.4.1], it is possible for multiple devices to be assigned the same COM port number, resulting in a COM port conflict.

To check for a COM port conflict

1. Display the Advanced Settings of the COM port associated with your device:
 - (a) Open the Windows Device Manager (**devmgmt.msc**).
 - (b) Locate the COM port for your device in the *Ports* subtree.
 - (c) Right-click the port and select *Properties* to display the Port Properties window.
 - (d) Select the *Port Settings* tab, and click the *Advanced...* button to display the Advanced Settings window.
2. In the Advanced Settings window, the *COM Port Number* box displays the port's name (which indicates the port number) – e.g., "COM10"; click the drop-down arrow to see a list of all COM ports in the system, and look for ports with the same name as that assigned to your device.
3. If there is a conflict – i.e., if another port has the same name – reassign the COM port for your device [J.4.3].

J.4.3 Reassigning the COM Port for Your Device

If your device's COM port number conflicts with that of another COM port, you can reassign the COM port for your device, using either the Windows Device Manager [J.4.3.1] or the DriverCore **change_port** utility [J.4.3.2].

J.4.3.1 Reassigning the COM Port using the Device Manager

To reassign the COM port for your device using the Windows Device Manager

1. In the Device Manager, display the Advanced Settings of the COM port associated with your device (see explanation in section J.4.2, step #1).
2. Click the drop-down arrow of the **COM Port Number** box; in the COM ports list, select a COM port that is not in use, and click **OK**.
3. Disable and re-enable the port in the Device Manager:
 - (a) Right-click the port and select **Disable**.
Windows disables the port.
 - (b) Right-click the port and select **Enable**.
Windows enables the port and activates the driver.

J.4.3.2 Reassigning the COM Port using the DriverCore change_port Utility

DriverCore includes a command-line utility for reassigning COM port numbers – **change_port**. This utility (provided in the DriverCore **tools** directory) assigns new COM port numbers using the operating system's COM port database [J.4.1].

To reassign the COM port for your device using the **change_port** utility

1. Open a command prompt.
2. Change directory to the DriverCore **tools** directory:
cd <path to the DriverCore directory>\tools.
For example: **cd c:\DriverCore\tools.**
3. Run **change_port** with the COM port number currently assigned to your device: **change_port <current port number>**.
For example, if your device is currently associated with COM3, run **change_port 3**.
The **change_port** utility assigns a different COM port for your device.

J.5 Debugging the DriverCore Drivers

The DriverCore drivers can be configured to print debug messages to the operating system log; these messages can be captured and logged using a kernel debugger, such as DebugView [J.5.2] or WinDbg.

To log debug messages from the DriverCore drivers

1. Set the desired debug level for the drivers [J.5.1].
2. Disconnect your device from the host machine, or disable it in the Device Manager.
3. Establish a debug session using your preferred kernel debugger (e.g., DebugView [J.5.2]).
4. Reconnect the device to the host machine, or enable it in the Device Manager (depending on the method used in step #2).
5. Reproduce the problem, then save and analyze the debug log.
You can send the log to Jungo for assistance [J.6].

J.5.1 Setting the Debug Level

The **DebugLevel** registry parameter controls the types of debug information that the DriverCore drivers send to the operating system log.

DriverCore defines the following **DebugLevel** values, which determine the types of debug messages (if any) that will be sent by the driver:

Value	Name	Description
0	Release	No debug messages
1	Error	Error messages only
2	Info	Error messages and basic information messages
3	Warning	Error, basic information, and warning messages
4	Verbose	All debug messages – error, basic information, warning, and additional information messages

The default debug level is 2 (*Info*).

When sending debug information to Jungo [J.6], set the debug level to 4 (*Verbose*).

NOTE

Processing debug messages impacts computer performance, especially when using the higher debug levels. When distributing the drivers to third parties, it is recommended that you set the debug level to 0 (*Release*).

You can set the debug level either by editing the driver INF files [J.5.1.1], or by editing the Windows registry directly [J.5.1.2].

NOTE

DriverCore defines a separate **DebugLevel** parameter for each driver; therefore, the debug levels of the enumerator and child drivers must be set independently.

J.5.1.1 Setting the Debug Level in the INF Files

You can set the debug levels for your DriverCore drivers by editing the driver INF files, and then reinstalling the drivers:

1. Close any applications that were used to communicate with your device, to ensure that there are no open handles to the device.
2. Edit the **DebugLevel** in the enumerator and/or child INF files [E].
3. Uninstall [4.3] and reinstall [4.1] the drivers to apply the changes.

J.5.1.2 Setting the Debug Level in the Registry

You can set the debug levels for your DriverCore drivers by editing the Windows registry directly, and then reloading the drivers.

CAUTION!

Be careful when editing the registry, since a mistake could damage the operating system. Microsoft recommends backing up the registry before editing it. For more information about working with the registry, see <http://support.microsoft.com/kb/256986>.

To set the drivers' debug levels directly in the registry

1. Close any applications that were used to communicate with your device, to ensure that there are no open handles to the device.
2. Disconnect the device from the host machine, or disable the device in the Device Manager. As a result, the drivers are unloaded.
3. Start the Windows registry editor (**regedit**).

4. For each of the drivers for which you want to set the debug level, edit the debug level value using the registry editor:
 - Navigate to the driver parameters keys:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\<driver name>\Parameters`
Note: Substitute **<driver name>**, above, with the name of your enumerator or child driver.
 - Double-click the **DebugLevel** key, to display the Edit Value dialogue.
 - Enter the desired debug level value, then click **OK**.
5. Reconnect the device to the host machine, or enable it in the Device Manager (depending on the method used for unloading the drivers – see step #2).
The drivers are loaded with the new debug settings.

J.5.2 Capturing Debug Information using DebugView

The Windows DebugView utility enables you to capture debug information provided by Windows applications and drivers.

NOTE

DebugView is available for free from
<http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>.

To capture debug information using DebugView

1. Start DebugView.
2. From the **Capture** menu, select all options except for **Log Boot**.
3. From the **Options** menu, select the **WIN32 PIDs** option.
4. From the **Computer** menu, select the **local machine** option.
5. Enable driver debugging and reload the drivers [J.5].
6. When you are finished capturing debug information, save the DebugView output.

J.6 Getting Help

If you are unable to get the DriverCore drivers to work properly, after reading the documentation and the troubleshooting guidelines in the previous sections, contact Jungo[K] for assistance.

To help us provide you with the most efficient support, include the following information with your request:

1. The name and version of the operating system
2. The CPU (processor) brand and model
3. The CPU architecture (e.g., 32 or 64 bit)
4. A screen capture of the Windows Device Manager (**devmgmt.msc**) device tree, set to **View > Devices by connection** [J.3.1], with the subtrees expanded
5. A screen capture of your device's hardware IDs, as displayed in the Device Manager [J.2]. For a composite device, attach a hardware-IDs screen capture for each of the device functions for which the DriverCore drivers failed to load. Note: The Device Manager usually displays an error indication ("!") next to problematic devices/functions.
6. If the Device Manager shows an error ("!") for the device and/or any of its functions – for each erroneous instance, attach a screen capture of the Properties window's **General** tab, which lists the error code.
7. The following log files, from the %WINDIR% directory (for example, C:\WINDOWS):
 - SetupAPI log file(s):
 - On Windows Vista: **setupapi.app.log** and **setupapi.dev.log**, from the **inf** subdirectory.
 - On older versions of Windows: **setupapi.log**
 - **setupact.log**
 - **Wdf01007Inst.log**
8. If the enumerator and/or child driver fail to load – reproduce the problem while using a kernel debugger [J.5], and attach a copy of the debug log
9. A USB analyzer capture file, containing the device descriptors of your device [J.3.3] (optional)

Appendix K

Contacting Jungo

For additional support and information, please contact us:

Web site: <http://www.jungo.com>

Email: sales@jungo.com

USA (toll free):

Phone: +1 877 514 0537

Fax: +1 877 514 0538

France (toll free):

Phone: +33 800 908 062

Worldwide:

Phone: +972 74 721 2121

Fax: +972 74 721 2122