Examples Guide: MSP430 USB API Stack

MSP430

1 Introduction

This document is intended for the person evaluating the MSP430 USB API stack. The API is accompanied by many examples that demonstrate its use; this document describes how to run those examples, and provides commentary on how they were written.

For detailed information on developing with the USB API, please see the MSP430 USB API Programmer's Guide, also located within the USB Developer's Package.

Table of Contents

1.1 Supported USB Device Classes

The API supports four USB device classes, summarized below. (Please see the Programmer's Guide for complete information.)

Table 1.   USB Device Classes Supported by the MSP430 USB API

Device Class	Description
Communications Device Class (CDC)	Produces a virtual COM port on the host
Human Interface Device (HID) Class	Traditional HID devices include mice and keyboards. The MSP430 USB API also defines a subclass called HID-Datapipe, which creates a UART-like free-form datastream on top of the HID interface.
Mass Storage Class (MSC)	A USB interface through which a storage volume can be mounted on the host
Personal HealthCare Device (PHDC)	Used with Continua healthcare devices

CDC, HID, and MSC cover the most common usages. PHDC is specific to Continua healthcare applications, and although it's supported by the USB API and Descriptor Tool, it is not documented within the USB Developers Package. If you need PHDC, please see the MSP430 Continua page.

A given USB device contains one or more *USB interfaces,* each of which adheres to a particular device class. A device containing more than one interface is said to be a *composite* USB device. Therefore, a device implementing a virtual COM port and a storage volume is a composite USB device, containing two USB interfaces: CDC and MSC.

1.2 Summary of the Examples

The examples are listed in the table below.

In USB, the vendor ID / product ID pair (VID/PID) is used to identify a unique USB product. See Sec. 2.6 for a discussion on how this affects using the examples.

Table 2. MSP430 USB API Application Examples, and PID Map

Number	Name	Example Type	USB VID/PID
C0	Simple sending, over CDC / virtual COM port	CDC	`0x2047 / 0x0300`
C1	Command-line interface with LED on/off/flash		
C2	Receive 1K data		
C3	Echo back to host		
C4	Packet protocol		
C5	High-bandwidth sending using `cdcSendDataWaitTilDone()`		
C6	Efficient sending using `cdcSendDataInBackground()`		
H0	Simple sending, over HID-Datapipe	HID-Datapipe	`0x2047 / 0x0301`
H1	Command-line interface with LED on/off/flash		
H2	Receive 1K data		
H3	Echo back to host		
H4	Packet protocol		
H5	High-bandwidth sending using `hidSendDataWaitTilDone()`		
H6	Efficient sending using `hidSendDataInBackground()`		
H7	Circular mouse	HID-Traditional	`0x2047 / 0x0309`
H8	Simple keyboard		`0x2047 / 0x0315`
M1	File-System Emulation (FSE)	MSC	`0x2047 / 0x0316`
M2	Interfacing with an SD-card, using FatFs		`0x2047 / 0x0317`
M3	Defining multiple LUNs		`0x2047 / 0x0318`
M4	Using double-buffering, for increased speed		`0x2047 / 0x0322`
M5	Implementing CD-ROM; cross-platform autorun		`0x2047 / 0x0323`
CH1	Composite CDC+HID Device; communicate between Terminal and HID Demo App	Composite	`0x2047 / 0x0302`
CC1	Composite CDC+CDC Device; communicate between two terminal apps		`0x2047 / 0x0313`
HH1	Composite HID+ HID Device; communicate between two HID Demo App Instances		`0x2047 / 0x0314`
CHM1	Composite CDC+HID+MSC, in which the MSC consists of two LUNs		`0x2047 / 0x0319`
CHM2	A SYSBIOS example that uses tasks. Composite CDC+HID+MSC, in which the MSC consists of two LUNs	SYSBIOS	`0x2047 / 0x0320`
CHM3	A SYSBIOS example that uses tasks. Composite CDC+HID+MSC, in which the MSC consists of two LUNs		`0x2047 / 0x0321`
G1	Improving interrupt latency on USB start/resume	General	`0x2047 / 0x0300`
	User experimentation area		`0x2047 / 0x03DF – 0x03FD`

2 Running the Examples

2.1 Obtaining Code Composer Studio or IAR Kickstart

The USB API stack and examples build and run on both the IAR and Code Composer Studio (CCS) environments for MSP430. Support for GCC is pending, but not yet available. See the Release Notes HTML file in the USB Developers Package for specific IAR/CCS version information.

IAR and CCS are both available in free, code-size-limited versions (8K and 16K, respectively, of object code). Simple applications having no MSC interface can be run on both free versions. Applications having an MSC interface (and thus a larger application to implement the storage volume) generally don't fit under 8K; for this reason, the free IAR Kickstart won't build them. Instead, the free version of CCS can be used, or a licensed version of either environment.

Be sure you're using the appropriate IAR/CCS version, for this version of the USB Developers Package. For this and other information specific to a given release, see the Release Notes within the USB Developers Package.

2.2 Hardware Support

Most of the examples can run on any hardware TI sells in the eStore, that supports a USB-equipped MSP430 device. This includes:

- F5529 LaunchPad (MSP-EXP430F5529LP)
- F5529 Experimenters Board ((MSP-EXP430F5529)
- Any FET target board for any USB-equipped MSP430 derivative (MSP-TS430RGC64USB, MSP-TS430PN80USB, MSP-TS430PZ100USB)

Most of the examples require no special configuration in the code, to adjust for hardware. The only required action is to select the appropriate MSP430 derivative in the project settings. (This is described in the next section.)

There are these exceptions:

- #M2-5: these require hardware with an SD-card socket. The F5529 Experimenters Board has this, and these examples are designed to run on that board.
- #H8: this requires pushbuttons. Every board above has buttons, but they're located on different pins. A *hal.h* file is provided in the example; select the appropriate board there. The list of options includes all the boards in the list above.

Every example has a hal.c/h file pair, where clocks and ports are initialized. These are the same on every example other than the exceptions above.

One hardware-specific resource is not configured in hal.c: the XT2 oscillator, on which a crystal/ or resonator is required for USB operation. This resource is owned by the USB API, and thus is configured within the *descriptors.h* file. This file is generated by the Descriptor Tool, and so it should be configured in the Tool. All the TI hardware listed above uses a 4MHz crystal on XT2.
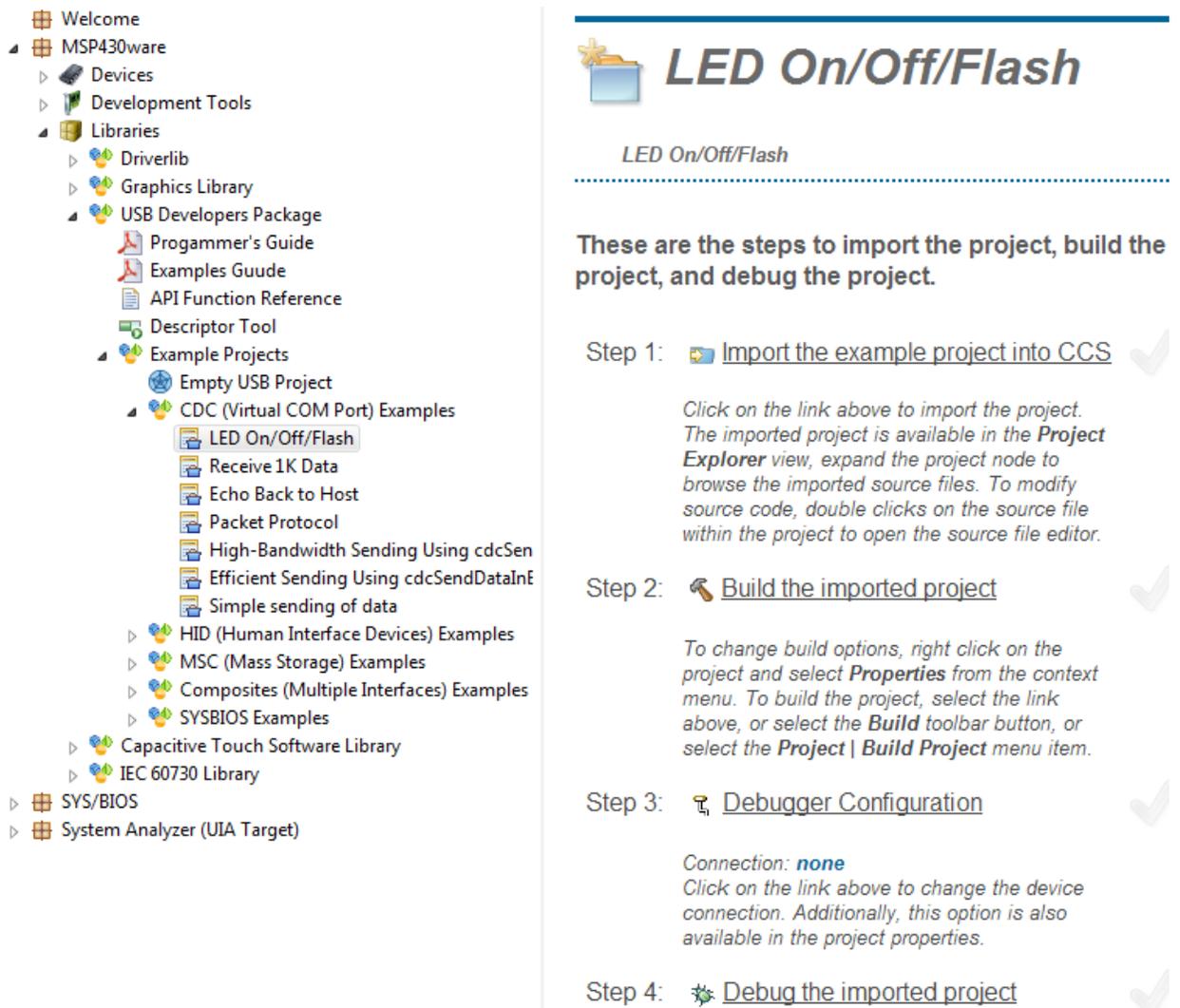
2.3   Exploring the USB Examples

There are two ways to explore the USB examples:

- The TI Resource Explorer, integrated into CCS
- Open the project workspace, which contains all the examples, into CCS/IAR

Either way provides a big-picture view that enables easy exploration of the examples.

2.3.1 Using the TI Resource Explorer in CCS

The TI Resource Explorer shows every element of the USB Developers Package, including every example. It also includes a step-by-step procedure that helps you build and download.

Figure 1.   Viewing the Examples in the TI Resource Explorer

To use the examples:

- In CCS, choose View → TI Resource Explorer.
- In the Resource Explorer's navigation tree, select Packages→MSP430Ware→Libraries→USB Developers Package
- The examples can be seen, as shown in the figure above. Click on the example, then follow the instructions on the right side of the pane.

Also note the *Empty USB Project* entry, underneath *Example Projects.* When the time comes, you can use this to help create your own USB project. The step-by-step instructions will walk you through using the USB Descriptor Tool to define your USB interfaces.

2.3.2 Opening the Examples Workspace Inside CCS/IAR

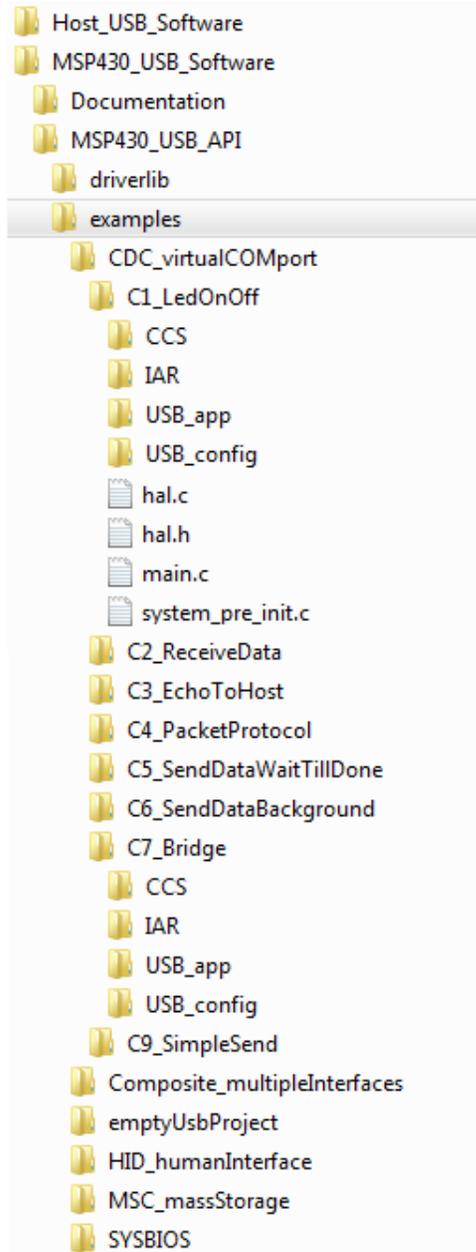For each example, projects are provided for both the IAR and CCS environments.

Figure 2.   Examples Directory

Instructions for opening these projects are in the following sub-sections.

Note that each example has a common structure, shown in the table below.

Table 3.   Example Directory Description

Directory Name	Description
\CCS	Contains a .projectspec file, which CCS uses to load this project
\IAR	Contains *.ewd/*.ewp files, which IAR uses to load this project
\USB_app	Files specific to this example, not including main.c or hal.c/h
\USB_config	Files related to the USB Descriptor Tool. **descriptors.c/h, usbisr.c:** code files that configure this example's USB interfaces and the descriptors it reports to the host. **INF file:** (if the device included a CDC interface) generated by the Tool, which you will need when installing the CDC interface onto a Windows PC ***.dat file:** stores the Descriptor Tool inputs that were used to generate this output. You can open this file with the Descriptor Tool.
hal.c/h	Code that initializes clocks and ports.
main.c	The example's main code.
system_pre_init.c	In CCS, executes immediately after the reset vector, prior to initializing RAM before the first line of main(). Stops the watchdog early. Used for examples defining large amounts of RAM variables, to ensure their initialization doesn't cause a device reset prior to the first line of main().

All the examples link to two directories above the \examples directory:

Table 4.   Linked Directories

Directory Name	Description
\USB_API	Contains the USB API stack itself
\driverlib	Contains a subset of the MSP430 driverlib library, used by the examples to access MSP430 peripheral modules other than USB

2.3.2.1 Opening the IAR Projects

The IAR projects are grouped within workspaces. There is one workspace per group of examples.

To use the examples with IAR, open the workspace file (*.eww) for the desired. After doing so, one of the projects in the workspace view will be highlighted in bold; this is the active project. (If a different example is desired, right-click on it and select "set as active".)

Open the "project options" for the active project:
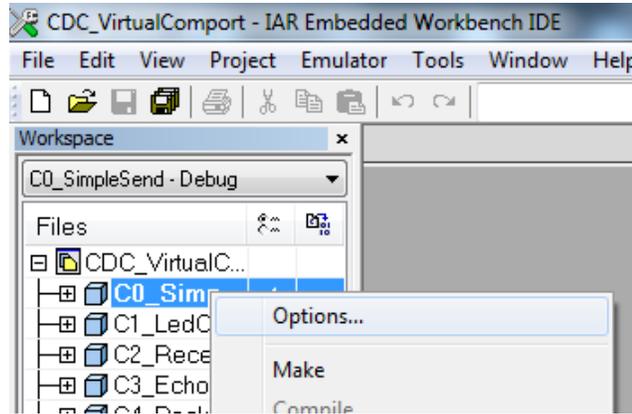
Figure 3.   Opening "Project Options" in IAR

In the view shown below, choose the MSP430 device derivative being used:
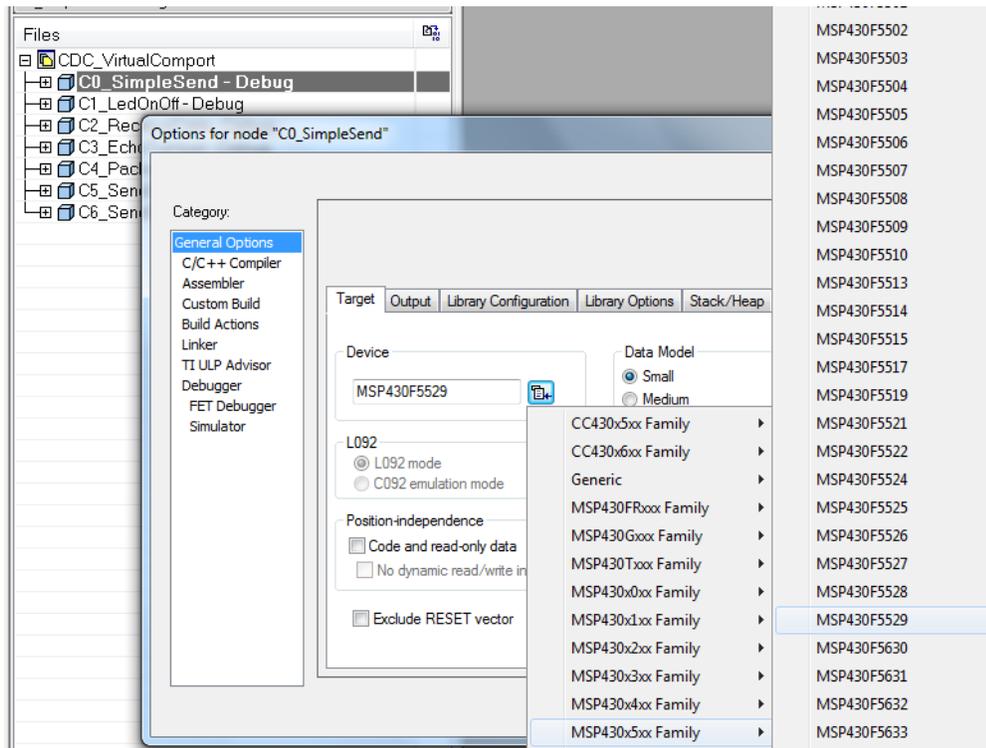
Figure 4.   Choosing the MSP430 Device Derivative in IAR

Then click "OK". The entire project is now automatically configured for the appropriate MSP430 device. It can be built.

If using example #H8_Keyboard, be sure to select the appropriate TI hardware in the file *hal.h*, as decribed in Sec. 2.2.

The example can now be run.

2.3.2.2 Opening the CCS Projects

The CCS projects are not grouped in workspaces. They need to be imported into a workspace of your choosing. In v4.0 and later of the USB Developers Package, the projects are defined by *.projectspec files, which contain the information CCS needs to import the project.

Open CCS, and choose Project → Open Existing CCS Eclipse Project. Browse to the \CCS directory of the example you wish to open. (This directory contains the .projectspec file.)
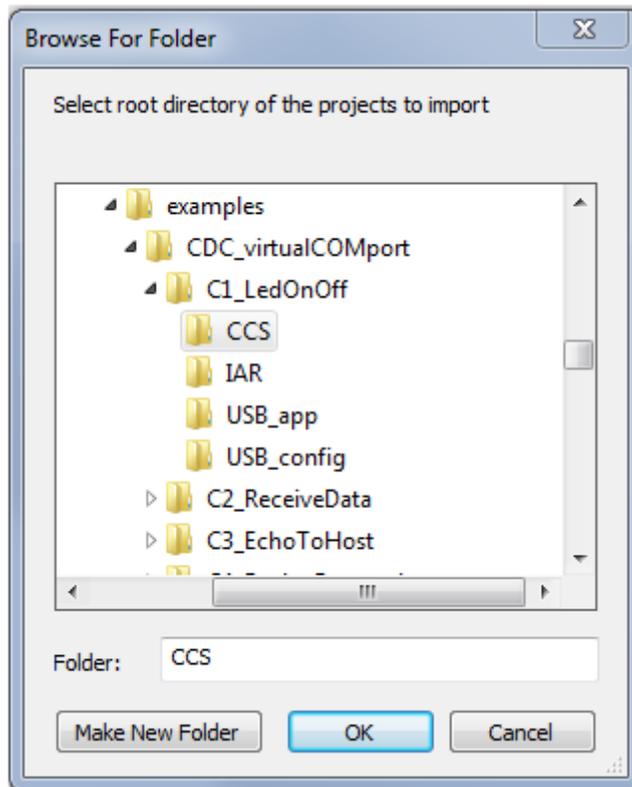
Figure 5. Importing the CCS Project (Step 1)

Press "OK". CCS now sees the project.
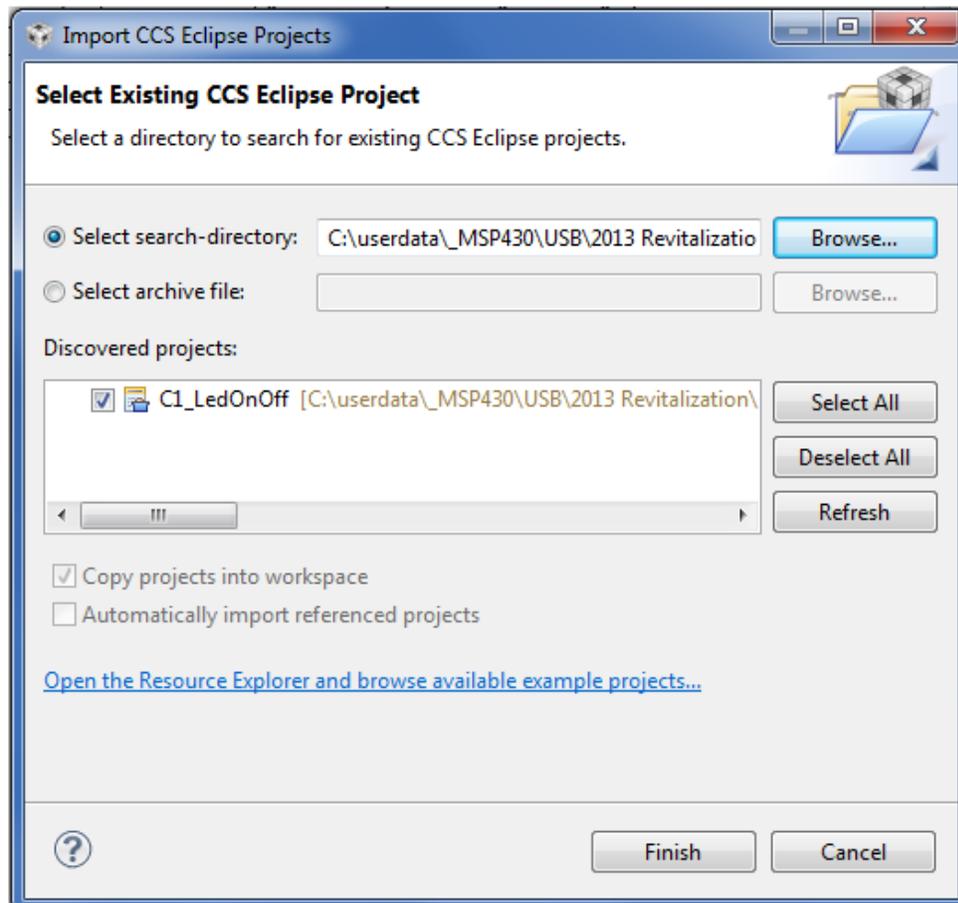
Figure 6.   Importing the CCS Project (Step 2)

Press "Finish". The project should appear in the *Project Explorer.*

To select the MSP430 device derivative being used, open the project's options, by right-clicking on the project, and selecting "Properties". Then select the "General" view, and select the appropriate "Variant" (MSP430 device derivative). Press "OK".
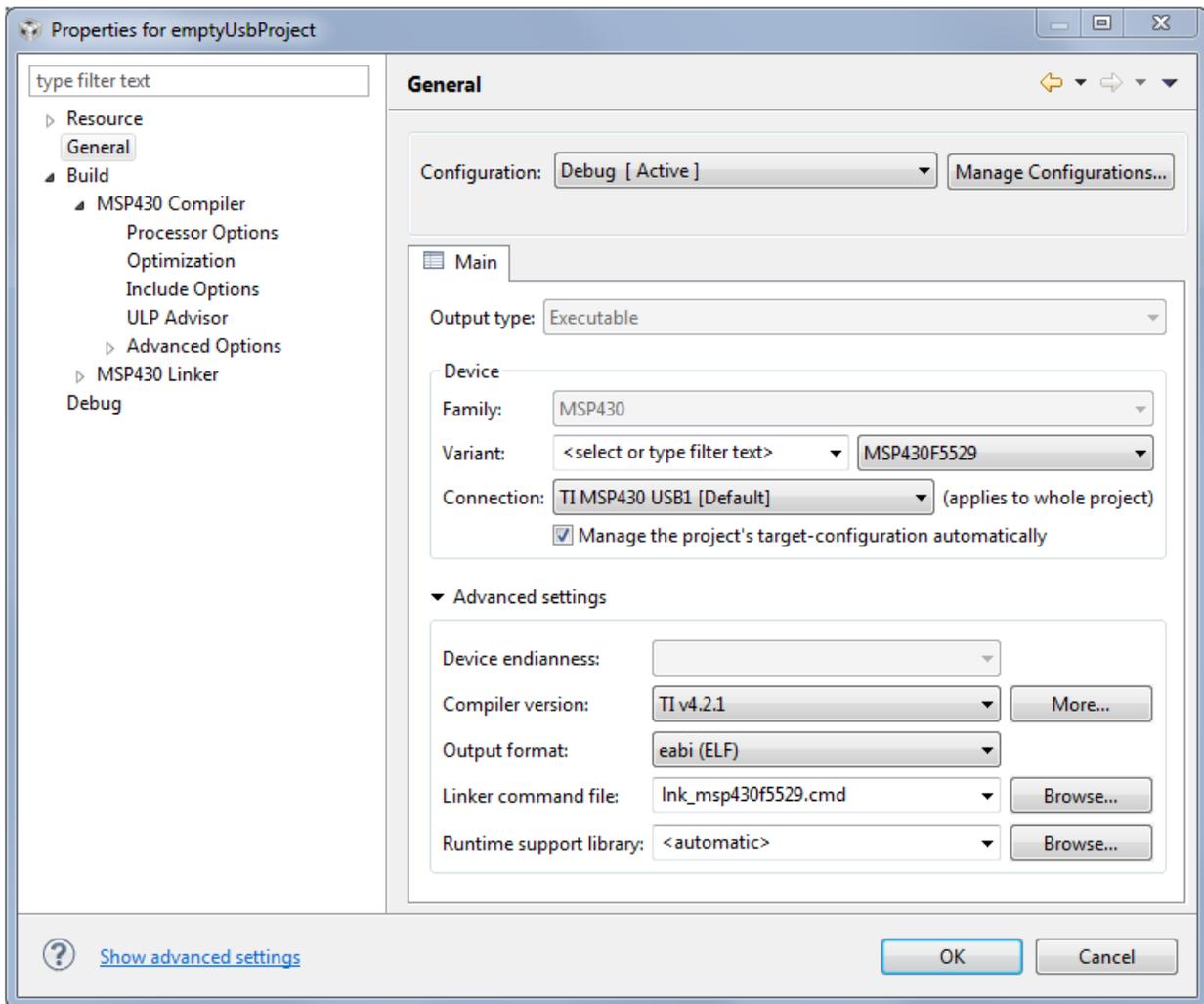
Figure 7. Choosing the MSP430 Device Derivative in CCS

The project is now fully configured for the new MSP430 derivative. Like with IAR, the API and application adapt to the device selection. It can be built

If using example #H8_Keyboard, be sure to select the appropriate TI hardware in the file *hal.h*, as decribed in Sec. 2.2.

The example can now be run.

2.4 Host PC Software to Interact with the USB Device

For CDC and HID-Datapipe, software on the host PC is required to interface to the USB device. MSC and traditional HID devices use software already present in Windows, Linux, and the MacOS.

2.4.1 Host Software for CDC Interfaces

CDC interfaces generate a COM port on the host. Therefore, the CDC examples are designed to interface with a general-purpose "terminal" application.

Every host OS platform (Windows/Linux/Mac) has several options for these; each may have different behavior, but serve the same basic function.

Since Hyperterminal was ubiquitous on Windows until recently, this guide uses Hyperterminal as an example. Another good terminal application available online for Windows is Docklight. PuTTY is an example terminal application for Linux.

Note that the baudrate, start/stop bits, and flow control settings on the host software don't matter. This COM port is a virtual one, and there is no actual UART on which to configure these settings. The host operating system will allow you to configure them, but it won't have any effect on these examples.

The first time a device with a CDC interface is attached to a Windows PC (meaning, the first time the PC has seen this device's VID/PID), a device installation process will be required. See Sec. 2.5 for more information about this. The rest of this section assumes this process has already completed successfully.

Once the CDC interface has been installed, the host application must open the COM port associated with the MSP430 device. This raises the question of how to identify the correct COM port? Every host platform has a way of observing the virtual COM ports software applications can access. In Windows, this is the Windows Device Manager, shown below.
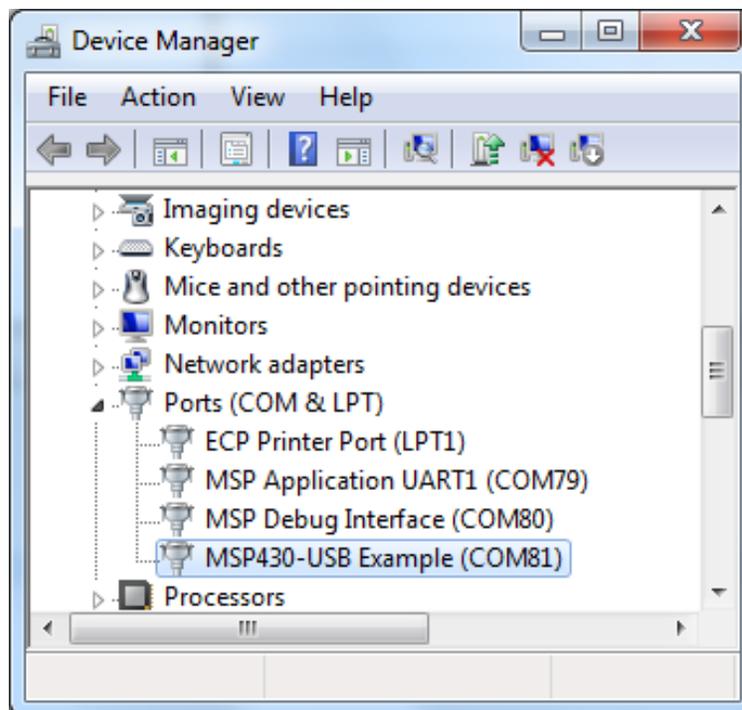
Figure 8. Identifying the COM Port Using the Windows Device Manager

The text label for COM ports comes from the INF file that was used to install it. For the MSP430 USB API, this is entered into the Descriptor Tool. The USB CDC examples are all listed as a MSP430-USB Example as shown above.

Alternatively, unplug/plug the device a few times, and see which device in the list disappears and re-appears.

2.4.2 Host Software for HID-Datapipe Interfaces: the Java HID Demo App

The USB Developer's Package includes a Java-based utility for HID-Datapipe called the *Java HID Demo App*. HID-Datapipe is a type of HID interface included in the MSP430 USB API, which implements a UART-like unformatted datastream over an HID interface.

The application's source and object code are located within the USB Developers Package, at:

```
\Host_USB_Software\Java_HID_Demo\
```

Figure 9.   HID Demo App Directory

For information on the host OS platforms supported, see the release_notes.html file.

The HID Demo App's functionality is very similar to that of terminal applications, except it uses HID-Datapipe as the underlying interface, instead of CDC.

Figure 10.   HID Demo App

To run the app, double-click on HidDemo.jar.

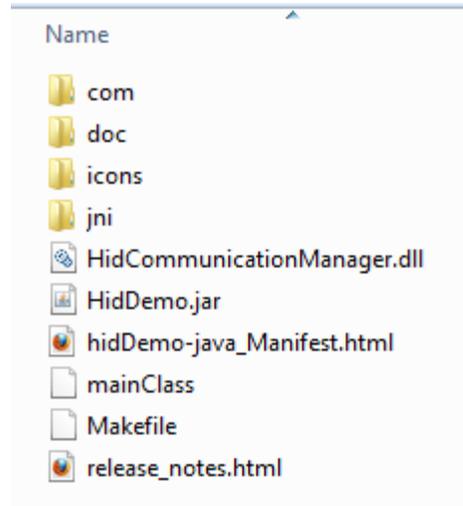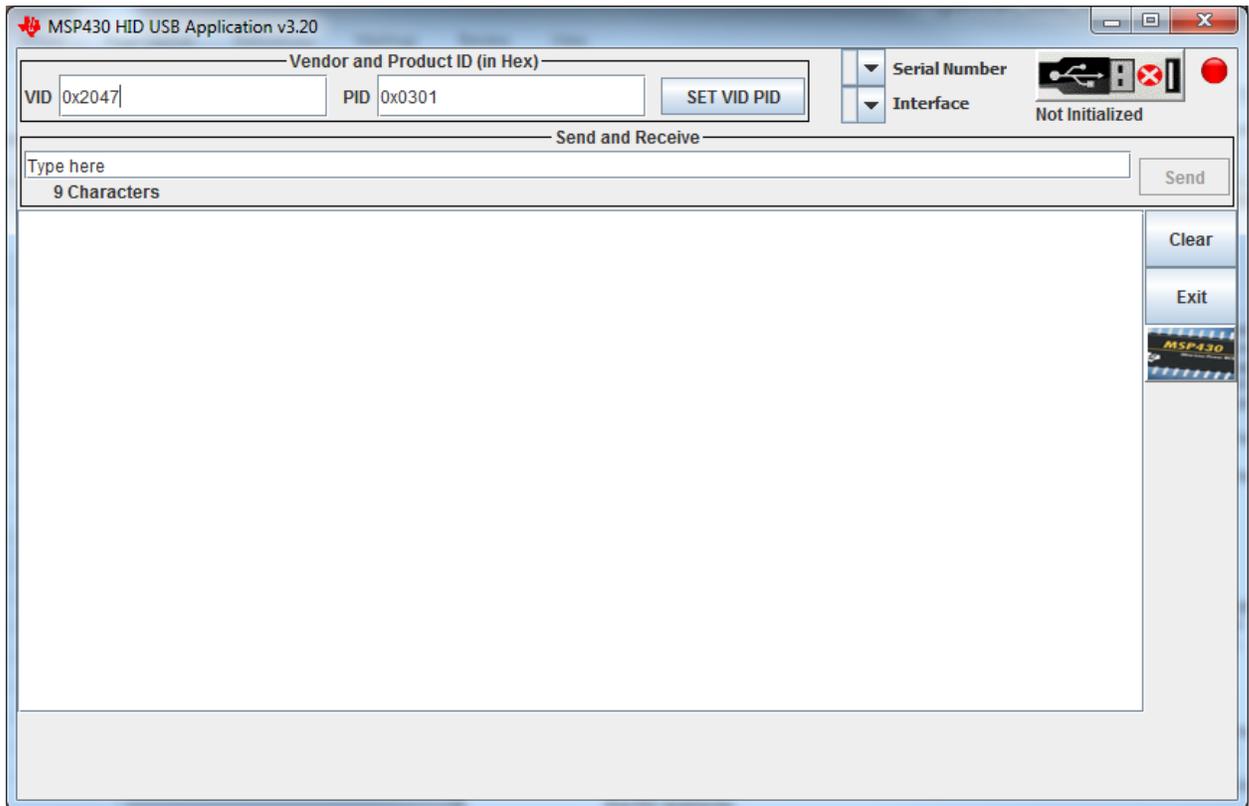The app can distinguish any HID interface on any USB device attached to the host. It first locates all devices sharing the same VID/PID pair; these are assumed to all have the same USB descriptor set, as multiple physical instances of a product would. The unique serial number of each of these devices is displayed. A serial number is unique to any physical USB device. So whereas the vendor ID (VID) identifies the device's vendor, and the product ID (PID) identifies a specific product, the serial number identifies a specific physical instance of that product.

Finally, each HID interface in each of those devices is displayed. The user can thus access any HID interface on the host.

After double-clicking on HidDemo.jar, the app is launched.
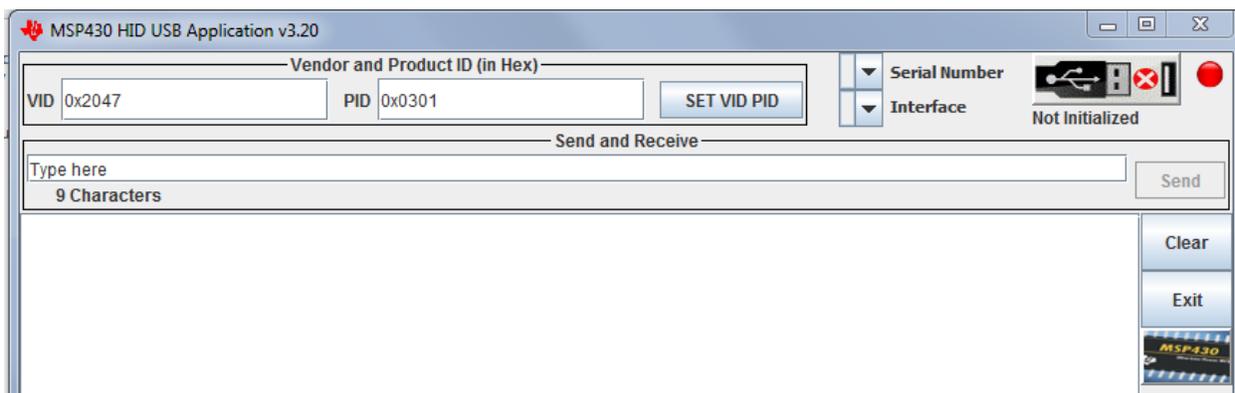
Figure 11.   Java HID Demo App

Enter the VID and PID for the USB device with which you want to interface. The USB examples have a VID of 0x2047, and the PIDs are shown in Table 2. (The single-interface HID examples all have a PID of 0x0301.)

After entering them, press "SET VID PID". This causes the Demo App to scan the USB devices attaches to this PC for a device matching this VID/PID. For each device found, a unique serial number is displayed in the "Serial Number" menu. For each entry in this menu, each HID interface on that device is listed in the "Interface" menu, in the order they were defined in the USB descriptors ("HID 0", "HID 1", etc.).

Once the specific interface has been selected, press the "Connect" button. This initiates a connection with the chosen HID interface.

The application can be opened in multiple instances, for interfacing with more than one HID interface.

If connection is not successful, try the following:
- Check the Windows Device Manager to ensure that the device successfully enumerated on the system as a HID device
- Ensure that the VID/PID selected in the Demo App matches the ones shown in Table 2. (The values in this table reflect what is shown in *descriptors.h, generated by the Descriptor Tool.*)
- Try pressing the "Set VID/PID" button again.

Once the connection is initialized, data can be sent to the device by entering text and pressing "Send". Data received by the application from the device at any time is displayed in the large text field. The receive window can be cleared with the "Clear" button.

In the CDC examples, strings are terminated with a return character, and the applications look for this. The HID Demo App's "Send" button doesn't automatically add a return character. Instead, end the string with a "!" character before pressing send. The HID examples look for this character as a means of terminating the string.

2.4.3 Host Support for Traditional HID Interfaces

Examples of traditional HID devices include mice and keyboards. The example set includes each of these.

With the mouse, the host operating system itself acts as the host "application". There is no need for additional host software.

With the keyboard, the host will require any application that receives text, like a text editor.

2.4.4 Host Support for MSC Interfaces

The host operating system recognizes and mounts any storage volume attached to it, whether the underlying bus is SATA, SCSI, USB, etc. Applications can then read/write files on the volume.

Therefore, it will do the same with the USB MSC examples. Some of the examples specifically use text files, for interaction with simple text editors (i.e., Windows' "Notepad").

2.5 CDC Interfaces on Windows: INF Files and Device Installation

On Linux and the MacOS, CDC interfaces load "silently", meaning no user action is required; a COM port simply becomes available. The same is true for HID and MSC interfaces on Windows, Linux, and the MacOS.

But the first time a particular device containing a CDC interface is attached to Windows, a "device installation process" is required. All Windows drivers require an INF file to associate the driver's binaries with a given device. For the HID and MSC drivers, these INF files are contained within the Windows installation. For the CDC driver, unfortunately it is not. This means it must be provided somehow by the user.

MSP430's USB Descriptor Tool generates the INF file, customized for the device's exact interface set and VID/PID. Each example contains a matching INF file, in the \USB_config directory.

Steps for installing this INF file on Windows XP, Windows 7, and Windows 8 are described in the following sub-sections. But first, the concept of driver signing needs to be addressed.

2.5.1 INF Signing

Microsoft requires device drivers to be *signed*. This requires submission to Microsoft.

Although the CDC driver binaries (usbser.sys) are native to Windows and are not changed by the OEM, the INF file contains the VID and PID for the vendor's product, which does change. Microsoft interprets this VID/PID change as a "custom driver", and requires it to be re-signed.

Since each OEM is intended to have a slightly different INF file (with its own VID/PID), it's difficult for TI to solve the problem for its customers.

If the INF file has not been signed, Windows XP and Windows 7 will still accept it, after the user approves the installation of an unsigned "driver". Windows 8 does not accept it, unless the user has taken special, unusual actions to allow it. This is described in Sec. 2.5.4.

The result of the signing process is a CAT file. Including this file in the same directory as the INF file, and referencing it within the INF file, will cause Windows to bypass asking the user to approve installation of an unsigned driver.

2.5.2 Installing a CDC Interface on Windows XP

When you build, run, and attach a CDC example for the first time, per the instructions in Sec. 2.3, WinXP begins a device installation process by displaying this dialog box:
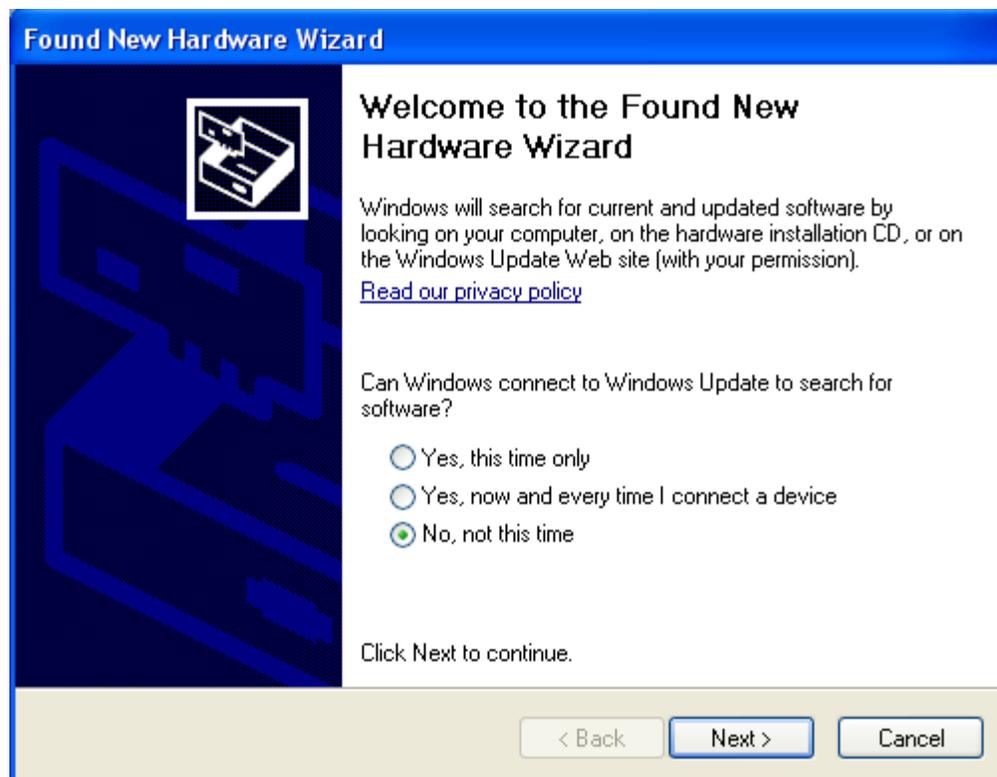

Figure 12.   Windows' Device Installation ("Found New Hardware") Box

Specifically, this happens the first time the host sees the device report a combination of VID, PID, and serial number that it hasn't encountered before.

Since the INF file has not been signed and registered with Microsoft, it is not available on Windows Update. Therefore, select "No, not this time", and press "Next".

This causes the following box to display:
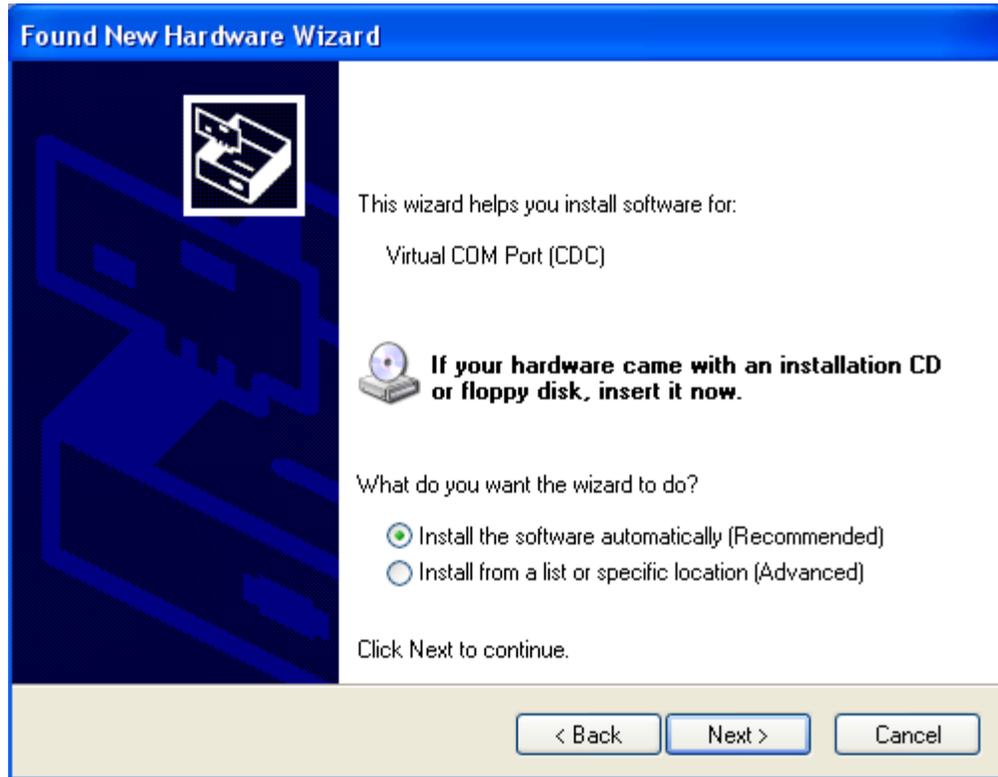
Figure 13.   Device Installation, Second Box

Perhaps the VID/PID are new (the same product), but the serial number is new (a different physical instance of that project). If so, then it has the INF file on the system already. Select "Install the software automatically (Recommended)", and press "Next". Windows will look in its history and find it, finishing the installation.

But if this VID/PID are new, then the INF isn't on the system. Select "Advanced", then press "Browse". This opens a "file open" box. Guide it to the USB example's directory, which contains an INF file loaded with the correct information for this example. (The "OK" button becomes enabled anytime a directory is selected that contains an INF file.)

All the single-interface CDC examples (#C0-C6) have the same VID/PID. They can do this, because their descriptor sets are identical. This means that after a successful device installation of any one of these, then operating a different example from this set -- on the same hardware -- will enumerate silently. The reason the hardware is relevant is that it will cause a different serial number to be reported. In that event, Windows will generate the box, but you can select the option "Install the software automatically (Recommended)".

Windows keeps track of previous installations in the system registry, using a hardware ID as an index, which is based on the VID/PID.

The details of this can be a little complicated. Just bear in mind these things:

- For the end user installing a single CDC device, the process is quite straightforward; they're only doing it once.
- Windows Update can automate the procedure
- A good installer can eliminate this procedure.
- INF files are only necessary on Windows machines, and only for CDC. On a Linux/Mac machine, CDC devices enumerate silently – no installation process required. On a Windows machine, enumerating an HID or Mass Storage device also happens silently.
- Even though an INF file must be distributed to the end user, the actual driver binaries are already in every Windows installation.

2.5.3 Installing a CDC Interface on Windows 7

When you build and run a CDC example for the first time, per the instructions in Sec. 2.3, Win7 does not display a dialog box as WinXP does. Instead, a bubble in the system tray may appear, indicating that drivers were not installed.

To install the INF file, launch the Device Manager. (See Appendix A for how to do this.)

The USB CDC examples appear in the Device Manager as "MSP430-USB Example". If a driver isn't properly installed, it appears with a yellow "!" on its icon, colloquially referred to as a "yellow bang".

Right click on that item:
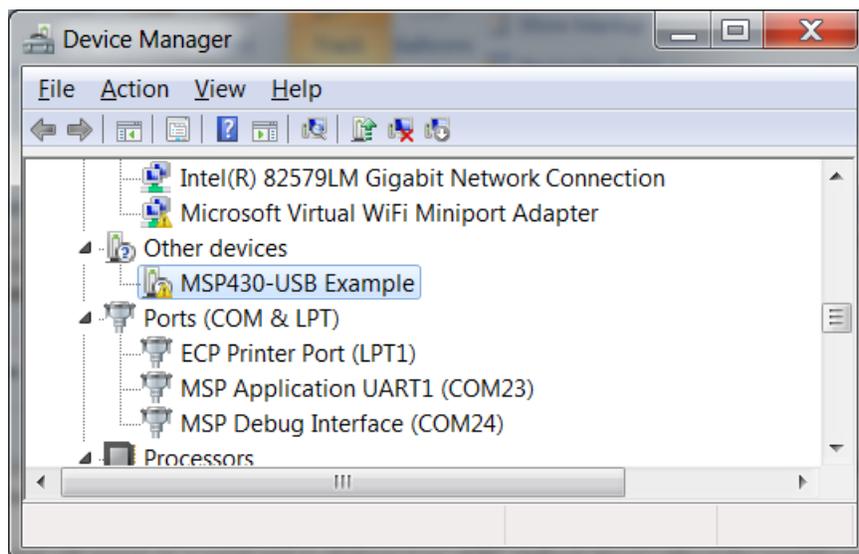
Figure 14. Device Manager Showing CDC Driver Not Installed

From the resulting contextual menu, select 'Update Driver Software'.
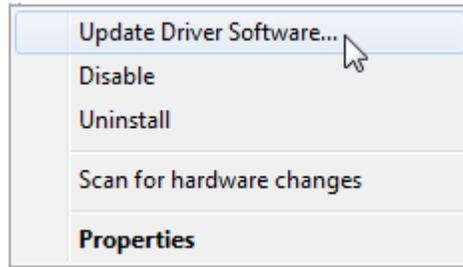
Figure 15. Update Driver Software

Choose "Browse My Computer for Driver Software":
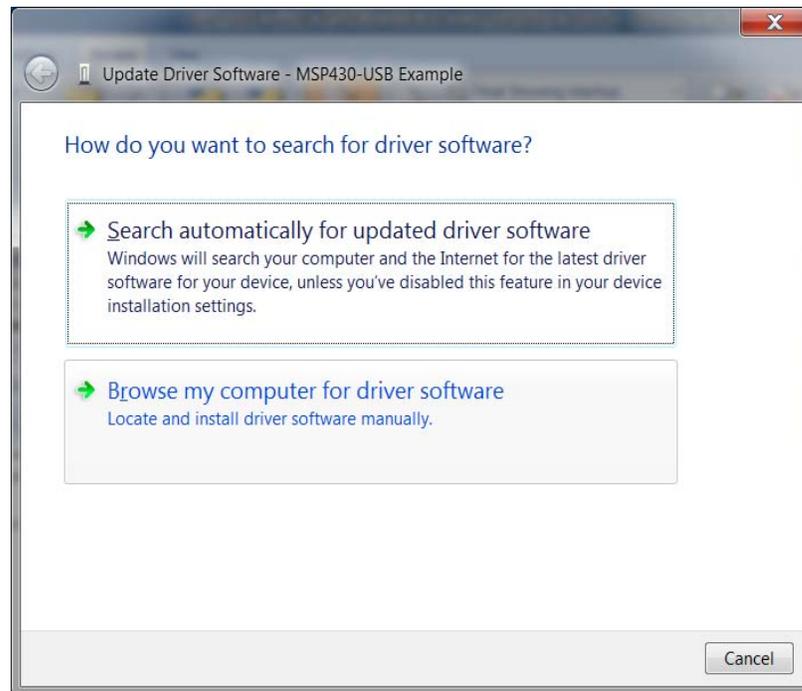
Figure 16. Update Driver Software

Navigate to the \USB_config directory for the CDC example in question. Be sure to choose the one for the right example.
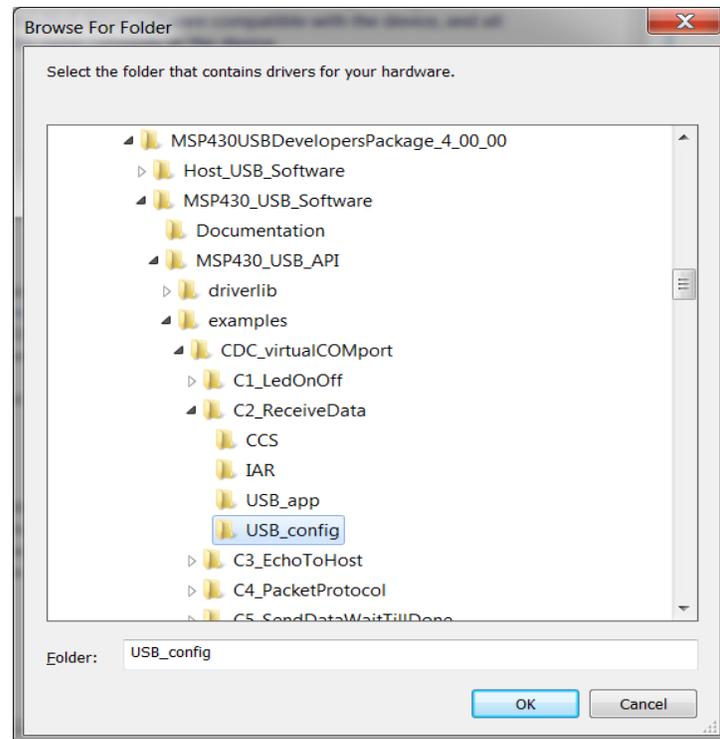
Figure 17.   Project Content with USB_Config

Then, click "OK". The Device Manager should now show the device without the "yellow bang" icon, with an assigned COM port number.
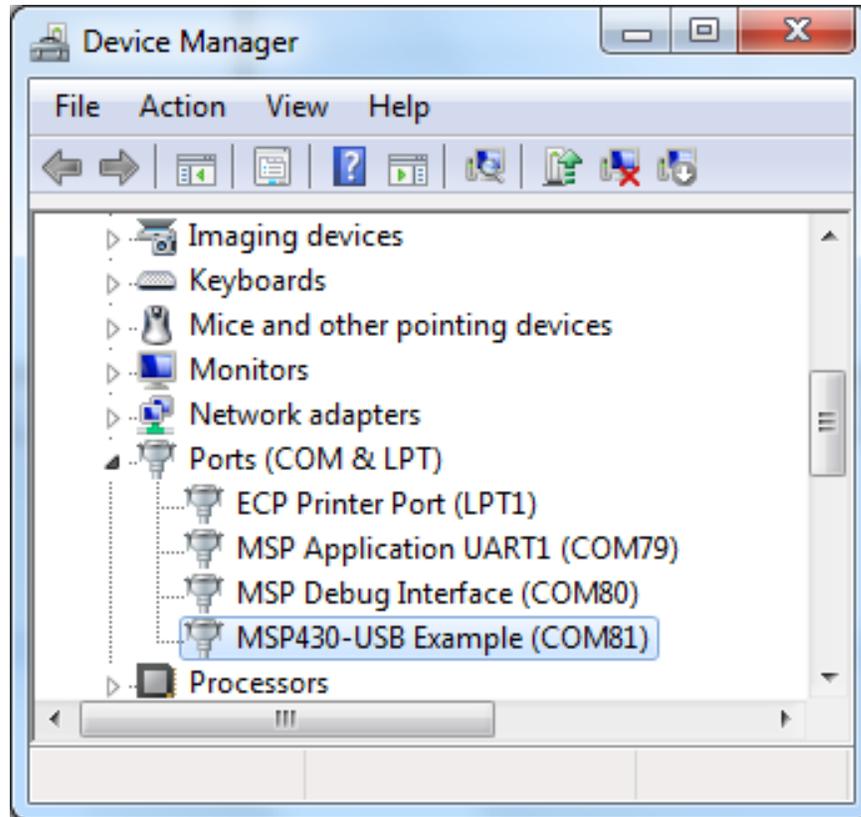
Figure 18. Device Manager with a Fully-Installed CDC Interface

NOTE: If the Device Manager lists the COM port as anything greater than 9, then verify that the general purpose "terminal" application like HyperTerminal being used will support COM ports greater than 9.

2.5.4 Installing a CDC Interface on Windows 8

Unlike WinXP/Win7, which allow installation of an unsigned driver (or INF file) if the user grants permission, Win8 will not, under normal conditions. However, in implementing this restriction, Microsoft still needed to provide a means of turning it off, to allow driver development. This is because a driver can't be signed until after it's completed.

Under normal conditions, attaching a device with a CDC interface, for which no signed INF was pre-installed, results in this dialog box.
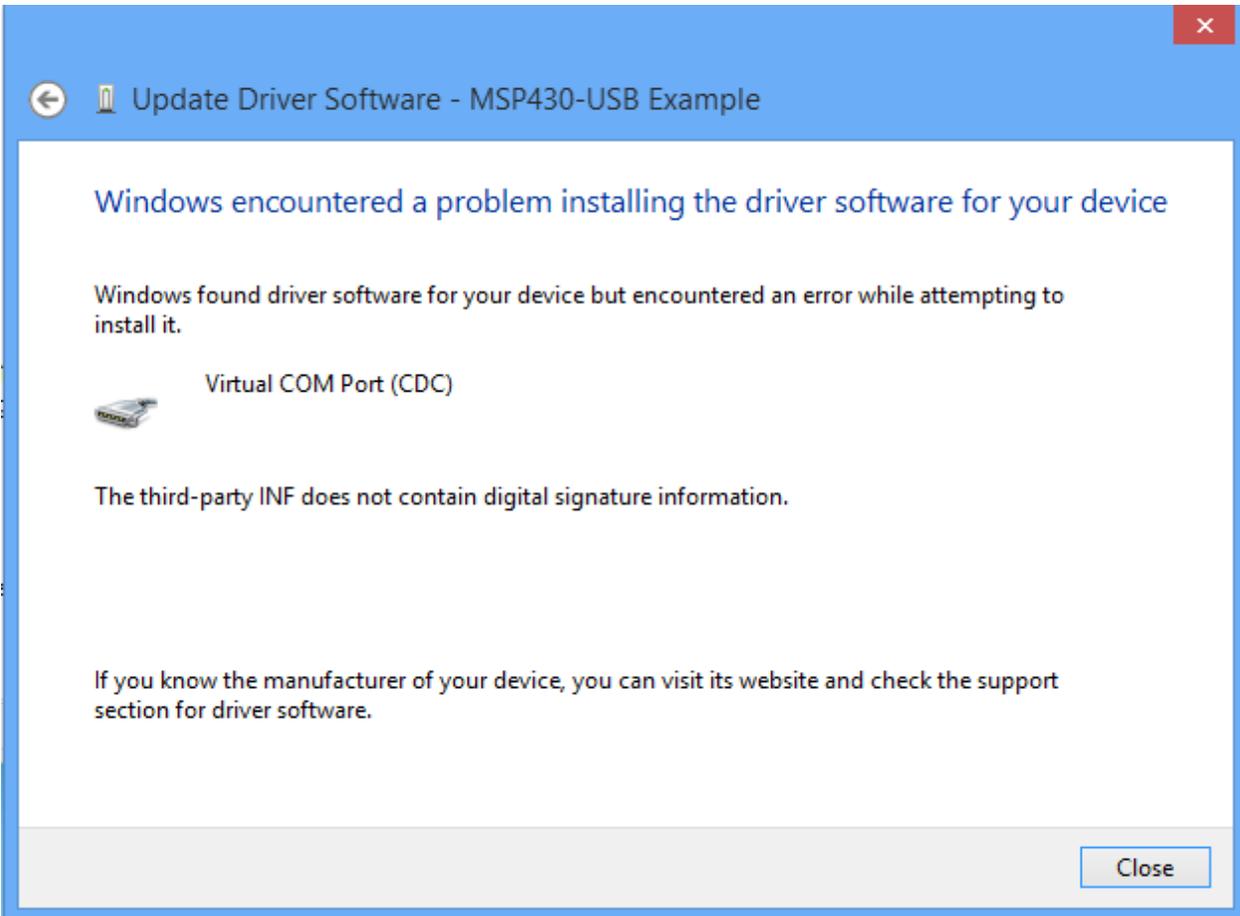
Figure 19. Win8's Response to Attaching a CDC Device, without Pre-Installed Signed INF

The solution Microsoft provides is a relatively hidden means of disabling enforcement of driver signage. The disabling is effective until the next reboot. The device/INF can be installed during that boot session, and then during subsequent boot sessions, Win8 will see the installed driver and not generate the dialog box shown above.

To enable this mode, press Win+R to bring up the "Run" window. Type "shutdown.exe /r /o /f /t 00" (without the ""). Press "OK".
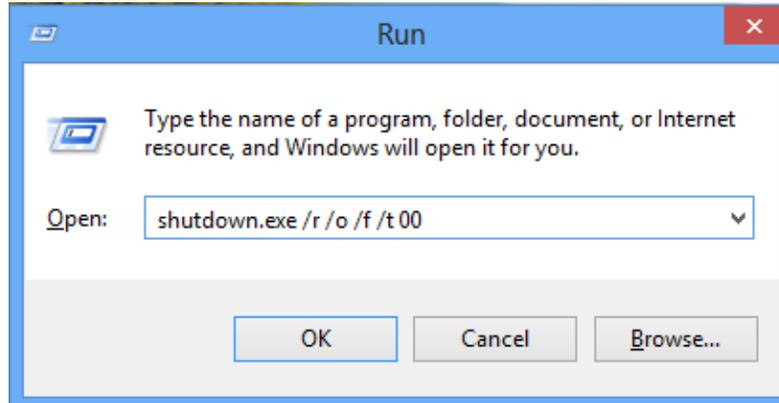
Figure 20. Disabling Win8 Driver Enforcement: Step #1

A reboot will occur, and the advanced startup menu displayed.
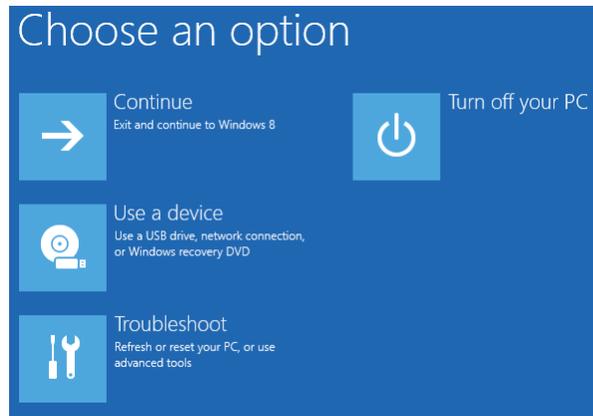
Figure 21. Disabling Win8 Driver Enforcement: Step #2

Click "Troubleshoot", then "Advanced Options", then "Startup Settings".
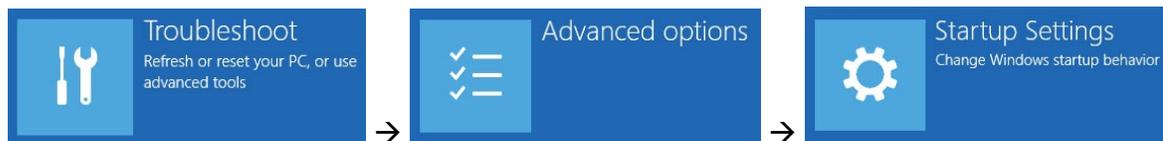
Figure 22. Disabling Win8 Driver Enforcement: Step #3

Finally, click on "Restart" in the bottom right corner, and wait for the reboot.

Figure 23.   Disabling Win8 Driver Enforcement: Step #4

When the computer restarts, select the option "Disable driver signature enforcement".
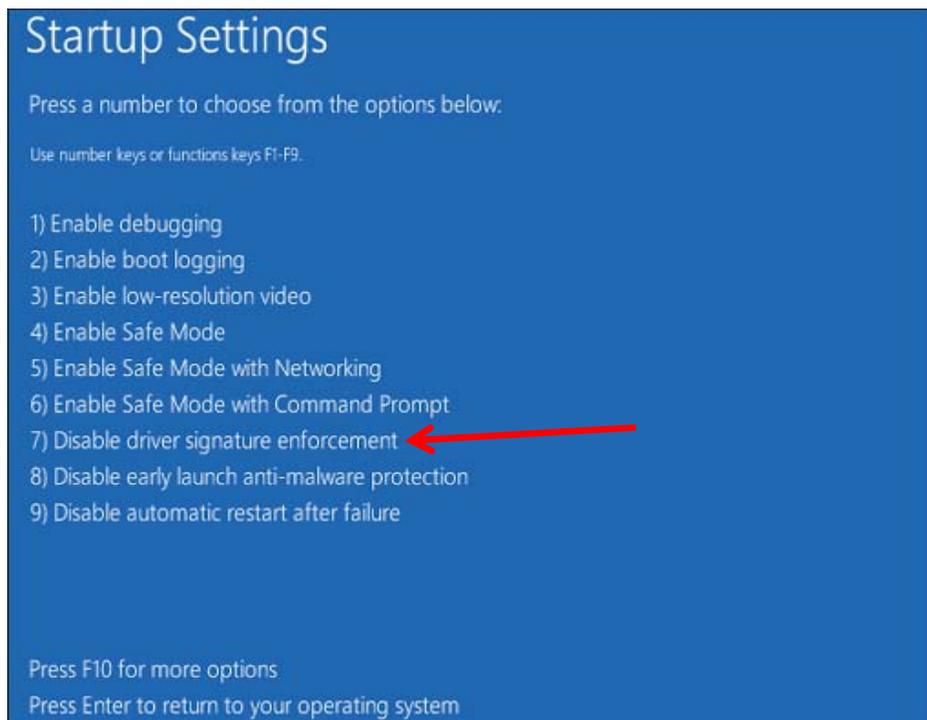
Figure 24.   Disabling Win8 Driver Enforcement: Step #5

When it finally boots into the operating system you can manually install the unsigned driver. You'll see the following message; just select "Install this driver software anyway.

Figure 25. Installing the Driver After Disabling Enforcement

The disabling lasts for one boot session. In the next re-boot, enforcement of driver signage will resume.

2.6   Take Care if Changing the VIDs/PIDs

Note that when running the examples, it's best to not change the VIDs/PIDs (configured in descriptors.h via the Descriptor Tool). This allows the USB host to properly keep track of the device information associated with each VID/PID. If the host PC encounters a VID/PID it's seen previously, it assumes the descriptor set hasn't changed. If it has in fact changed, the host will get confused.

In the event you decide to create your own USB programs, a group of PIDs is provided for you to assign to them. This is the "User experimentation area", at the bottom of the table. If you change USB descriptors during your development, be sure to use a new PID, to ensure the host saves new information for that VID/PID. (For a complete discussion on how USB VIDs/PIDs work, please see the Programmer's Guide.)

In each example, the VID is the one owned by TI MSP430: 0x2047.

Some PIDs are shared by multiple examples These are examples that all share the same set of USB interfaces and descriptor set. For example, all single-interface CDC examples use PID 0x0300.

Other examples have unique PIDs. Each HID-Traditional example has a unique PID, because each has a different HID report format. Similarly, the MSC examples have unique PIDs; this time, it isn't as much because of different USB descriptors, but because the host may choose to record information about the device's storage volume, which is different between the examples.

If there is any doubt, it is always safest to provide a unique VID/PID to the host, from the "User experimentation area". This is a set of 30 PIDs that neither TI nor anyone to whom we license our VID (via our VID-sharing program; see http://www.ti.com/msp430usb) will ever use for a product. As such, there should be no risk that these PIDs have been encountered by any host machine unless its owner/developer was the one who caused it.

3 Example Descriptions

3.1 General Instructions for Running Examples

This section contains instructions that are general to any example containing the stated interface type: CDC, HID, or MSC.

Beyond these general instructions, the rest of this section starting with Sec. 3.2 contain instructions specific to each individual example.

3.1.1 CDC Examples

The following steps apply to any example containing a CDC interface, unless otherwise noted.

1) Follow the instructions in Sec. 2.3 to select the target MSP430 derivative, build, and run the example project.

2) Connect the device to the USB. (It's also OK to connect, and then run the example.) You may wish to turn up the volume on your PC, to hear the tones indicating successful enumeration.

3) If necessary, follow the instructions in Sec 2.5 to install the CDC interface. CDC interfaces always enumerate silently on Linux/Mac hosts.

4) Identify the COM port the host OS has associated with the CDC interface. On Windows, this can be done with the Device Manager (see Appendix A). Look for a new device appearing under "Ports", named "MSP430-USB Example" or "Virtual COM Port (CDC)". In the figure below, the COM port is COM25.
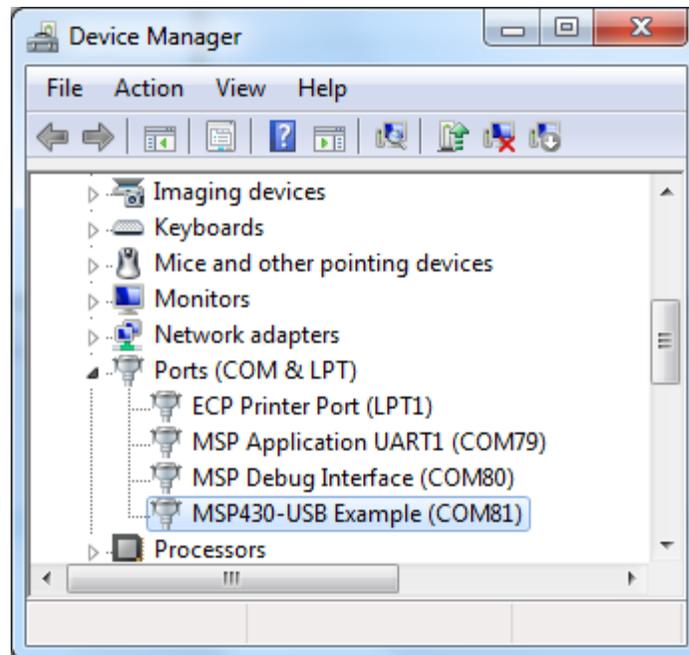
Figure 26. Identifying the COM Port Using the Windows Device Manager

5) Run a terminal application of your choice – for example, Hyperterminal, Docklight, or PuTTy - - and open the identified COM port in the Device Manager. The baudrate and other COM port configuration settings do not matter, since virtual COM ports aren't based on physical UART interfaces.

6) Follow the instructions for this specific example.

When using terminal applications like Hyperterminal with USB-based virtual COM ports, it will soon be noticed that they're not very tolerant of surprise removal of the USB cable. This is because COM ports were originally designed to work with permanently-attached hardware UART interfaces, not "plug and play" devices that can be detached at any time

To account for this, after removing the USB cable and attempting a re-connect, follow this order:
- Close the port within the terminal app
- Connect the USB device to the PC
- Now, re-open the terminal's connection to the port

In other words, the COM port should be closed when the USB device enumerates. The port can then be opened normally.

3.1.2 HID-Datapipe Examples

The following steps apply to any example containing a HID-Datapipe interface, unless otherwise noted.

1) Follow the instructions in Sec. 2.3 to select the target MSP430 derivative, build, and run the example project.

2) Connect the device to the USB. (It's also OK to connect, and then run the example.) You may wish to turn up the volume on your PC, to hear the tones indicating successful enumeration.

3) You may wish to verify that the interfaces installed properly. On Windows, you can launch the Device Manager, and locate the new device appearing at attachment under "Human Interface Devices". (See Appendix A.)

4) Run the Java HID Demo App, per the instructions in Sec. 2.4.2Use the instructions there to direct the app to this example's VID/PID.

5) Follow the instructions for this specific example.

3.1.3 Traditional HID Examples

Traditional HID interfaces (as opposed to HID-Datapipe) include mice and keyboards. A mouse example and keyboard example are both provided.

The following steps apply to any example containing a HID-Datapipe interface, unless otherwise noted.

1) Follow the instructions in Sec. 2.3 to select the target MSP430 derivative, build, and run the example project.

2) Connect the device to the USB. (It's also OK to connect, and then run the example.) You may wish to turn up the volume on your PC, to hear the tones indicating successful enumeration.

3) You may wish to verify that the interfaces installed properly. On Windows, you can launch the Device Manager, and locate the new device appearing at attachment under "Human Interface Devices". (See Appendix A.) It will also be viewable under "Mice and other pointing devices" and "Keyboards", depending on the example.

4) Follow the instructions for this specific example.

3.1.4 MSC Examples

The following steps apply to any example containing an MSC interface, unless otherwise noted.

1) Follow the instructions in Sec. 2.3 to select the target MSP430 derivative, build, and run the example project.

2) Connect the device to the USB. (It's also OK to connect, and then run the example.) You may wish to turn up the volume on your PC, to hear the tones indicating successful enumeration.

3) A new volume should appear on the system; for example, on Windows, it should appear under "My Computer".

4) You may wish to verify that the interfaces installed properly. On Windows, you can launch the Device Manager, and locate the new Mass Storage device appearing under "USB Controllers". After the volume mounts, it will be viewable under "Disk Drives". (See Appendix A.)

5) Follow the instructions for this specific example.

3.2 Single-Interface CDC Examples

3.2.1 Example #C0_simpleSend

3.2.1.1 Running It

The purpose of this example is to show the simplest possible exchange of data over USB. It doesn't check for the presence of the bus, or take special action to handle surprise removals.

The application maintains a real-time clock (RTC). Every second, if a USB host is present, it sends the time over the USB to a terminal application. If the USB host is not present, no data is sent.

Build and run the example, and then open the COM port, per the instructions in Sec. 3.1.1. The time should be reported every second.
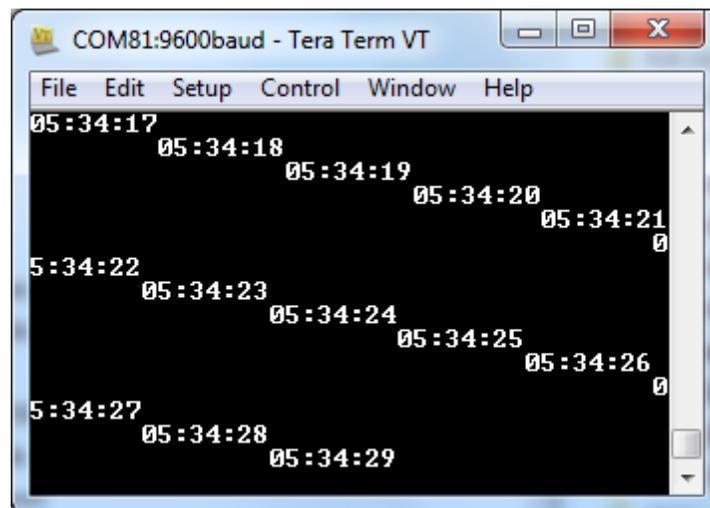
3.2.1.2 Implementation Comments

Unlike many of the other examples, the suggested switch() framework isn't used. As a result, the nature of this device doesn't change depending on whether a USB host is present or not. It is primarily, and always, a real-time clock. If the host happens to be there, it receives a copy of the time.

`cdcSendDataInBackground()` is used to send the data. This function starts the send operation and then immediately returns, allowing data to be sent while execution continues past this call. It does check to ensure a previous send operation isn't still underway; if so, it waits for a limited number of retries before giving up.

Compare this example to #H0_simpleSend, which is essentially identical except for the use of `hidSendDataInBackground()` rather than `cdcSendDataInBackground()`. The only difference is the underlying USB device class used.

3.2.2 Example #C1_LedOnOff

3.2.2.1 Running It

This example implements a simple command-line interface, where the command is ended by pressing 'return'. It accepts four commands. NOTE: The commands are case sensitive, and they must have spaces as shown.

- LED ON
- LED OFF
- LED TOGGLE - SLOW
- LED TOGGLE – FAST

After opening the COM port, you can enter these commands and watch the LED respond.

This example uses an LED. All the TI hardware boards shown in Sec. 2.2 have an LED on the location this example uses. If using different hardware, be sure to manually configure the LED's I/O in the code.

Some terminal applications display text locally as it's typed. Some, like Hyperterminal, don't. By default, this example echoes back text as it's typed. If using a terminal app that automatically echoes, the echo coming from the USB device can be eliminated by simply commenting out the line that does this.

The following is an example screen shot using PuTTY terminal application:
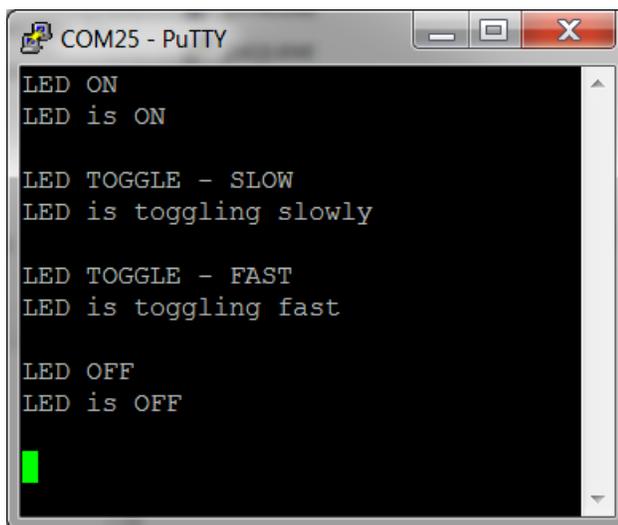
Figure 27.   C1_LedOnOff Example Screen Shot

3.2.2.2 Implementation Comments

This example uses the `cdcReceiveDataInBuffer()` construct function described in the Programmer's Guide. This function is advantageous in this example, because with a return-delimited command interface, there's no way to predict the exact number of bytes that will be received from the host. Characters are handled as they arrive; or if no characters arrive, this application will stay in LPM0 indefinitely.

Because of this piece-wise approach, each newly-arrived group of characters needs to be concatenated to a master string, and each group is searched for a return character that indicates the end of the string.

This example uses `cdcSendDataInBackground()` to send the response data. `cdcSendDataWaitTilDone()` could have been used as well, with no practical difference in the application.

3.2.3 Example #C2_ReceiveData

3.2.3.1 Running It

This example implements a device whose only purpose is to receive a 1K chunk of data from the host. It begins by prompting the user to press any key. When the user does so, it asks for 1K of data. Any data received after that point will count toward the 1K (1024 byte) goal. When 1K has been received, the program thanks the user, and the process repeats.

A text file with 1K of data is included in the example's directory. After opening the COM port, this file can be sent from Hyperterminal using the "Send text file" command. (Note that Hyperterminal and some other terminal applications use 1-byte packets for this function, and therefore the transfer may be slow.)

3.2.4 Implementation Comments

This application receives data in two different ways. It uses `USBCDC_handleDataReceived()` to wake up the main loop out of LPM0. The main loop then enters a clause that prepares to receive 1K of data, where it simply calls `USBCDC_receiveData()` to begin a receive operation for 1024 bytes.

Notice that execution resumes immediately after the receive operation is started, and doesn't wait for it to finish. When the operation does finish, a call to `USBCDC_handleReceiveCompleted()` will be generated. Like `USBCDC_handleDataReceived()`, this handler sets a flag and wakes the main loop. This time the main loop enters a clause that thanks the user and puts the application back in the "Press any key" state. The operation repeats as before.

3.2.5 Example #C3_EchoToHost

3.2.5.1 Running It

This example simply echoes back characters it receives from the host.

Open a COM port to the device, and begin typing. If the terminal application has the built-in echo feature disabled, typing characters into it only causes them to be sent; not displayed locally. This application causes typing in Hyperterminal to feel like typing into any other PC application – characters get displayed. If the echo feature is enabled, then each character will appear twice.
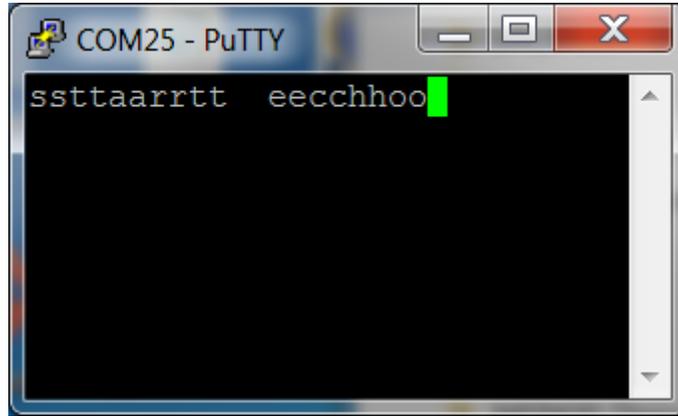
Figure 28.   C3_EchoToHost Example Screen Shot

3.2.5.2   Implementation Comments

Since there's no way to predict how many bytes will come -- or when – `cdcReceiveDataInBuffer()` is used. Most of the time, the application is in LPM0. When data is received, the application wakes and echoes the characters back.

3.2.6   Example #C4_PacketProtocol

3.2.6.1   Running It

This application emulates a simple packet protocol that receives packets like the following:
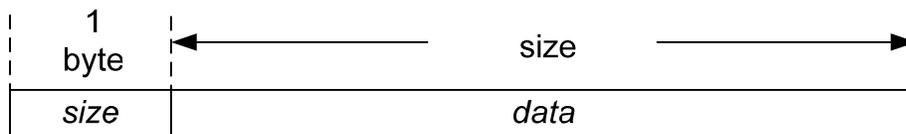
Figure 29.   Packet Format Implemented in this Example

Establish a connection with the device using the terminal application. No text is initially displayed. Type one digit (a number between 1-9) to enter the `size` value. Then, press any set of keys, for a total of `size` times. For example, type 3, and then type "abc". The application responds by indicating it has received the packet, and waits for another.

Note: This application assumes the size byte is received in a USB packet by itself. This is what happens when typing text into Hyperterminal, for example. If using a terminal application that gives control of this to the user, be sure to send the data separately from the size byte.
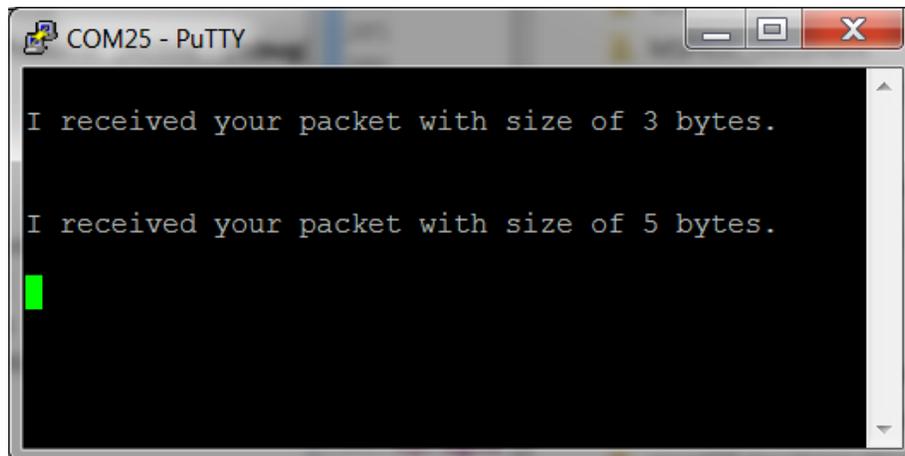
Figure 30. C4_PacketProtocol Example Screen Shot

3.2.6.2 Implementation Comments

This example begins a fixed-size receive operation for a single byte, since it's known that all "packets" in this protocol start with a one-byte size field. Then, this field is used to determine the size of the next fixed-size receive operation. When that operation completes, it displays text indicating as such.

3.2.7 Example #C5_SendDataWaitTillDone

3.2.7.1 Running It

The example implements large data transfer using `cdcSendDataWaitTilDone()`, which doesn't allow execution to proceed until all the data has been sent. It prompts for any key to be pressed, and when this happens, the application sends a large amount of data to the host.

3.2.7.2 Implementation Comments

At the beginning of execution, the application fills a buffer with characters that can be clearly displayed in the terminal application (no control characters).

As with some of the other examples, it uses the `USBCDC_handleDataReceived()` event to see the keypress, and then rejects the data received. This clears the USB endpoint buffer for the next keypress.

`cdcSendDataWaitTilDone()` is used, but `cdcSendDataInBackground()` could easily have been used instead. Indeed, in a real application, `cdcSendDataInBackground()` could raise efficiency and/or increase bandwidth.

No timeout value is used on the `sendData` calls. Because of this, if the COM port isn't opened on the PC, the call to `cdcSendDataWaitTilDone()` will wait forever for the "Press any key" send to complete (or, until the bus becomes unavailable through detaching from the host or being suspended).

Keep in mind all the factors that can affect bandwidth, as discussed in the API Programmer's Guide within the USB Developers Package. If using Hyperterminal, bandwidth will be slow because it uses only a single byte per USB packet.

3.2.8 Example #C6_SendDataBackground

3.2.8.1 Running It

The example shows how to implement efficient, high-bandwidth sending using background operations. It prompts for any key to be pressed, and when this happens, the application sends a large amount of data to the host.

Although this example demonstrates the technique, the bandwidth in this example will not vary significantly from that of `cdcSendDataWaitTilDone()`. This is because the example isn't performing any other large operations that need CPU time, and because the bus usually isn't loaded down. Also, with many terminal applications, data is also significantly slowed by the fact that they don't send/receive data using large USB packet sizes. This increases overhead.

3.2.8.2 Implementation Comments

At the beginning of execution, the application fills two buffers, `bufferX` and `bufferY`, with characters that can be clearly displayed in the terminal application (no control characters).

At the beginning of the main loop, it repeatedly sends "Press any key" to the host. In Hyperterminal, this appears as a fixed string. When any data is sent from the host, the API makes a call to `USBCDC_handleDataReceived()`. The actual data received doesn't matter; it is only used to start the transmission. The event sets a flag indicating received data, and this changes the flow within main(). The data remains in the USB endpoint buffer until the sending is complete, at which point it is rejected so that the process may begin again.

When the key is pressed, the application alternately sends `bufferX` and `bufferY`. They're alternately sent until *rounds* expires.

By itself, this approach doesn't provide any functionality that couldn't have been accomplished by a single call to `cdcSendDataWaitTilDone()` with a buffer the size of `bufferX` + `bufferY`. However, other code can now be inserted into the application that will be executed simultaneously with the send operations, increasing efficiency. Code flow is also more fluid with this approach, because execution won't be locked up in a polling loop after the send, as would have happened with `cdcSendDataWaitTilDone()`.

3.3 Single-Interface HID-Datapipe Examples

Examples #H0-H6 use the HID-Datapipe interface. Examples #H7 and #H8 use traditional HID interfaces.

Notice that the coding of the HID-Datapipe examples is highly symmetrical with that of the CDC interfaces.

3.3.1 Example #H0_simpleSend

3.3.1.1 Running It

The purpose of this example is to show the simplest possible exchange of data over USB. It doesn't check for the presence of the bus, or take special action to handle surprise removals.

The application maintains a real-time clock (RTC). Every second, if a USB host is present, it sends the time over the USB to a terminal application. If the USB host is not present, no data is sent.

Build and run the example, and then open a connection to the device using the Java HID Demo App, per the instructions in Sec. 3.1.2. The time should be reported every second.

3.3.1.2 Implementation Comments

Unlike many of the other examples, the suggested switch() framework isn't used. As a result, the nature of this device doesn't change depending on whether a USB host is present or not. It is primarily, and always, a real-time clock. If the host happens to be there, it receives a copy of the time.

`hidSendDataInBackground()` is used to send the data. This function starts the send operation and then immediately returns, allowing data to be sent while execution continues past this call. It does check to ensure a previous send operation isn't still underway; if so, it waits for a limited number of retries before giving up.

Compare this example to #C0_simpleSend, which is essentially identical except for the use of `cdcSendDataInBackground()` rather than `hidSendDataInBackground()`. The only difference is the underlying USB device class used.

3.3.2 Example #H1_LedOnOff

3.3.2.1 Running It

This example implements a simple command-line interface, where the command is ended by pressing 'return'. It accepts four "commands": NOTE: The commands are case sensitive, and they must have spaces as shown.

- LED ON!
- LED OFF!
- LED TOGGLE – SLOW!
- LED TOGGLE – FAST!

(Recall that when using the HID-Datapipe examples, a "!" character serves the same purpose as a return character did in the CDC example, in terminating a string.)

After opening a connection to the device with the HID Demo App, you can enter these commands and watch the LED respond.

This example uses an LED. All the TI hardware boards shown in Sec. 2.2 have an LED on the location this example uses. If using different hardware, be sure to manually configure the LED's I/O in the code.

3.3.2.2 Implementation Comments

This example uses the `hidReceiveDataInBuffer()` construct function described in the Programmer's Guide. This function is advantageous in this example, because with a return-delimited command interface, there's no way to predict the exact number of bytes that will be received from the host. Characters are handled as they arrive; or if no characters arrive, this application will stay in LPM0 indefinitely.

Because of this piece-wise approach, each newly-arrived group of characters needs to be concatonated to a master string, and each group is searched for a return character that indicates the end of the string.

This example uses `hidSendDataInBackground()` to send the response data. `hidSendDataWaitTilDone()` could have been used as well, with no practical difference in the application.

3.3.3 Example #H2_ReceiveData

3.3.3.1 Running It

This example implements a device whose only purpose is to receive a 1K chunk of data from the host.

Establish a connection with the HID Demo App. The program prompts the user to press any key. When the user does so, it asks for 1K of data. Any data received after that point will count toward the 1K (1024 byte) goal. When 1K has been received, the program thanks the user, and the process repeats.

The HID Demo App can send the full 1024 bytes at one time. A text file with 1K characters is provided, which can be copied/pasted into the HID Demo App.

3.3.3.2 Implementation Comments

This application receives data in two different ways. It uses `USBHID_handleDataReceived()` to wake up the main loop out of LPM0. The main loop then enters a clause that prepares to receive 1K of data, where it simply calls `USBHID_receiveData()` to begin a receive operation for 1024 bytes.

Notice that execution resumes immediately after the receive operation is started, and doesn't wait for it to finish. When the operation does finish, a call to `USBHID_handleReceiveCompleted()` will be generated. Like `USBHID_handleDataReceived()`, this handler sets a flag and wakes the main loop. This time the main loop enters a clause that thanks the user and puts the application back in the "Press any key" state. The operation repeats as before.

3.3.4 Example #H3_EchoToHost

3.3.4.1 Running It

This example simply echoes back characters it receives from the host. Establish a connection with the HID Demo App, and begin sending data.

3.3.4.2 Implementation Comments

Since there's no way to predict how many bytes will come -- or when – `hidReceiveDataInBuffer()` is used. Most of the time, the application is in LPM0. When data is received, the application wakes and echoes the characters back.

3.3.5 Example #H4_PacketProtocol

3.3.5.1 Running It

This application emulates a simple packet protocol that receives packets like the following:

1 byte	size
size	*data*

Establish a connection with the HID Demo App. No text is initially displayed. Send a single digit, a number between 1-9. Then, send that number of bytes. For example, send '3', and then send "abc". The application responds by indicating it has received the packet, and waits for another.

Note: the way this application is written, it assumes the size byte is received in a USB packet by itself, with no data behind it. It won't work to send "3abc" in a single transmission.

3.3.5.2 Implementation Comments

This example begins a fixed-size receive operation for a single byte, since it's known that all "packets" in this protocol start with a one-byte size field. Then, this field is used to determine the size of the next fixed-size receive operation. When that operation completes, it displays text indicating as such.

3.3.6 Example #H5_SendDataWaitTillDone

3.3.6.1 Running It

The example implements large data transfer using `hidSendDataWaitTilDone()`, which doesn't allow execution to proceed until all the data has been sent. It prompts for any key to be pressed, and when this happens, the application sends a large amount of data to the host.

3.3.6.2 Implementation Comments

At the beginning of execution, the application fills a buffer with characters that can be clearly displayed in the HID Demo App (no control characters).

As with some of the other examples, it uses the `USBHID_handleDataReceived()` event to see the keypress, and then rejects the data received. This clears the USB endpoint buffer for the next keypress.

No timeout values are used on the `sendData` calls. Because of this, if the host application doesn't establish a connection with the device and begin polling for data, the call to `hidSendDataWaitTilDone()` will wait forever for the "Press any key" send to complete.

Because HID uses USB's *interrupt transfer* type, the datarate is very predictable and not affected by a loaded bus or busy host. Using the HID datapipe interface, 62 bytes of data are transferred every USB frame (every 1ms), which makes the datarate a consistent 62KB/sec. This is in contrast to the CDC protocol, which uses bulk transfers; bulk has the capacity to be much faster than interrupt transfers, and most of the time it is. However, since any traffic on the bus or loading on the host can delay bulk transfers, the bandwidth is potentially variable, and has the capacity to be stalled for long periods of time.

3.3.7 Example #H6_SendDataBackground

3.3.7.1 Running It

The example shows how to implement efficient sending using background operations. It prompts for any key to be pressed, and when this happens, the application sends a large amount of data to the host.

Although this example demonstrates the technique, the bandwidth in this example will not vary significantly from when `hidSendDataWaitTilDone()` is used. This is because the example isn't performing any other large operations that need CPU time, and because HID isn't susceptible to delays resulting from a loaded bus.

3.3.7.2 Implementation Comments

At the beginning of execution, the application fills two buffers, `bufferX` and `bufferY`, with characters that can be clearly displayed in the HID Demo App.

At the beginning of the main loop, it repeatedly sends "Press any key" to the host. When any data is sent from the host, the API makes a call to `USBHID_handleDataReceived()`. The actual data received doesn't matter; it is only used to start the transmission. The event sets a flag indicating received data, and this changes the flow within main(). The data remains in the USB endpoint buffer until the sending is complete, at which point it is rejected so that the process may begin again.

When the key is pressed, the application alternately sends `bufferX` and `bufferY`. They're alternately sent until *rounds* expires.

By itself, this approach doesn't provide any functionality that couldn't have been accomplished by a single call to `hidSendDataWaitTilDone()` with a buffer the size of `bufferX` + `bufferY`. However, other code can now be inserted into the application that will be executed simulatenously with the send operations, increasing efficiency. Code flow is also more fluid with this approach, because execution won't be locked up in a polling loop after the send, as would have happened with `hidSendDataWaitTilDone()`.

3.4 Single-Interface Traditional HID Examples

3.4.1 Example #H7_Mouse

3.4.1.1 Running It

This example functions as a mouse on the host. It causes the mouse pointer to move in a circular pattern on the screen. Simply build and run the example. To re-gain control of the mouse, unplug USB.

Unlike the HID-Datapipe examples, this one does not communicate with the HID Demo Application. Rather, the host operating system acts as the "application".

3.4.1.2 Implementation Comments

The application begins by configuring the timer for interrupts to occur with a period of approximately 1/60 second. During USB enumeration, the main loop spends most of its time in LPM0. The timer interrupts prompt it to send a report to the host containing mouse data, prior to returning to sleep.

A custom report descriptor is defined in *descriptors.c,* which was generated by the Descriptor Tool.

Figure 31.   Selection of Mouse Report in Descriptor Tool

When creating a traditional HID interface, the software developer is usually responsible for creating the report descriptor, and inserting it into the Descriptor Tool. But for mice/keyboards, the Tool has an option to automatically install an appropriate report format. The developer then needs to write application code that sends reports to it. This example shows how to do this.

The host recognizes this descriptor as a "mouse descriptor". It sees a top-level collection of type "Generic Desktop", and a usage of "Mouse". When it sees this, it assumes ownership of the device, interpreting the reports as mouse data for the pointer.

The example was written for hardware that doesn't have a motion sensor. Therefore, the motion data is fabricated, using a lookup table. The lookup table is stored in flash, and an index rotates through the table. The data in the table contains data producing a circular motion.

When the timer interrupt occurs, the ISR sets a flag and keeps the CPU awake after the ISR returns. This allows execution to resume from the LPM0 entry in main(). The flag is evaluated, and the report is built and sent. There isn't much need to check the return value from `USBHID_sendReport()`; this is because mouse data is soon outdated and will soon be replaced. If a report somehow fails to be sent, the application will soon be sending another one. Thus the application doesn't wait for report to get sent. ??are we updating this example to react to events instead of using a timer??

3.4.2 Example #H8_Keyboard

3.4.2.1 Running It

This example functions as a keyboard on the host. It assumes the hardware has two pushbutton switches. Once enumerated, pressing one of buttons causes a string of six characters – "msp430" -- to be "typed" at the PC's cursor, wherever that cursor is. If the other button is held down while this happens, it acts as a shift key, causing the characters to become "MSP$#)".

Attach the device, and open an application that can accept text, like a text editor. Then press the buttons.

3.4.2.2 Implementation Comments

A custom report descriptor is defined in *descriptors.c,* which was generated by the Descriptor Tool.
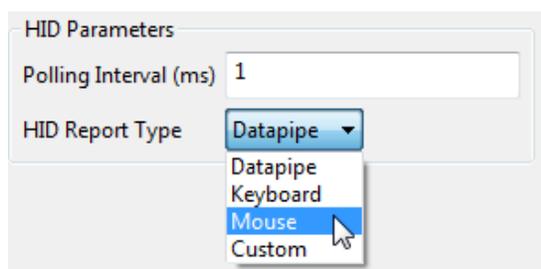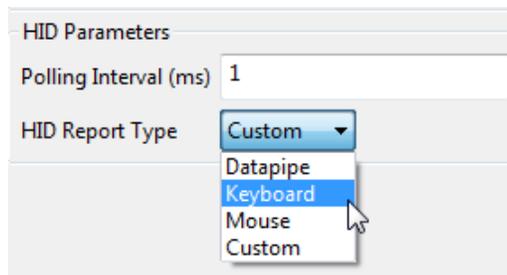

Figure 32.   Selection of Keyboard Report in Descriptor Tool

When creating a traditional HID interface, the software developer is usually responsible for creating the report descriptor, and inserting it into the Descriptor Tool. But for mice/keyboards, the Tool has an option to automatically install an appropriate report format. The developer then needs to write application code that sends reports to it. This example shows how to do this.

The host recognizes this descriptor as a "mouse descriptor". It sees a top-level collection of type "Generic Desktop", and a usage of "Keyboard". When it sees this, it assumes ownership of the device, interpreting the reports as keypress data.

3.5 Single-Interface MSC Examples

Note that most of the MSC examples use the FatFs open-source software for accessing FAT volumes. FatFs is beneficial when the MSP430 application needs the ability to parse and interpret the storage volume. In an application where this ability wasn't needed, like a simple flash drive, FatFs would not be needed; the application could simply carry out low-level block READ and WRITE commands from the host.

Example #M1 is slightly different; it uses file system "emulation".

3.5.1 Example #M1_FileSystemEmulation

This example demonstrates usage in which there is no file system software. Instead, a FAT volume is "emulated". Also, the entire contents of the volume are not stored inside the MSP430 flash; only eight blocks (512 x 10 = 5K) are emulated, excluding the Master Boot Record, File Allocation Table, and Root Directory.

Although the elimination of file system software has advantages (cost and less code space), and not having to store the entire volume saves a large amount of data space, this technique has limitations. Only eight small files, each less than 512 bytes, are supported (although it could probably be adapted for more). Also, care must be taken so that the end user doesn't do something to disrupt the volume. For example, the root directory might need to be made write-protected so that the user can't add additional files.

The reason this example is useful is that it supports a common usage model: using mass storage as a simple data exchange interface with the host. For example, the MCU application can log data gathered during normal operation into the file in storage; and then when attached to a host, the host can read it. The same can work in reverse.

3.5.1.1 Running It

Simply run the example and attach the device to the host. The volume appears on the system. For example, on a Windows PC, open "My Computer". A volume should appear entitled "Removable Disk".

Open the volume. There should be only one file in it: a text file entitled *data_log.txt.* Open this file using Windows' Notepad application. If the application had logged data and stored them in the FAT data, the host could then retrieve the data this way. Changes can be made and saved from Notepad as well.

3.5.1.2 Implementation Comments

As described at the beginning of this section, this example saves memory space and the cost of a file system, at the expense of only supporting one (or a few) file(s), and the extra effort to ensure the user can't disrupt the system through unintended actions.

The storage volume data (in mscFseData.c) is a system in itself. It is recommended to be very familiar with storage volumes before making significant changes. Having said this, there are a couple areas that can be fairly easily changed:

One is the short filename in the root directory (`Root_Dir[]`). This allows some flexibility in how the file gets presented to the host.

Another is the file attributes immediately beneath the filename in the root directory. With this, the file can be marked as 'hidden', 'read-only', etc. These attributes might be useful in hiding the file from the end user (while still be accessible to a custom application that knows what file to look for).

The Master Boot Record (`MBR[]`), file allocation table (`FAT[]`),root directory (`Root_Dir[]`) and data blocks are stored in flash. Since the intent is to prevent the host from adding files, there is no real need for it to be able to write to the root directory. Since the file is intended to not grow larger than one cluster, there is no real need to write to the FAT.

3.5.2 Example #M2_SDCardReader

This example demonstrates usage of the API with file system software. It includes an MSP430 port of the open-source "FatFs" software for the FAT file system.

This example requires hardware with an SD-card interfacelike the F5529 Experimenter's Board, available from TI's eStore. A boosterpack with an SD-card socket could also easily be developed for the F5529 Launchpad. (See Sec. 2.2 for information about these boards.)

This example implements a fully-functional SD-Card reader. It detects (and handles) live insertion and removal of the SD-Card, recovering gracefully. It's been tested on Windows XP/Vista/7, the Mac OS, and Ubuntu Linux. Although it's only been tested with the Experimenter's Board's "micro" SD-Card port, it should work with other forms of SD-Card as well. It works with "high-capacity" cards, as well as previous, smaller cards.

If adapting the example to other hardware, please note that some software adaptations were made with regards to detection of the SD-Card. The Experimenter's Board doesn't include the usual circuitry to detect insertion/removal with an I/O. Therefore, a software method was employed, described below. On alternative hardware, the hardware-based method might be desired.

3.5.2.1 Running It

Simply plug into the USB host. It should enumerate as a storage device, and the storage volume should be mounted on the system. On Windows, this means the volume will appear in "My Computer". If a card is present in the hardware, then the volume can be opened. If no card is present, then attempting to open it will probably result in a warning that no media is present. Insert one, and then re-attempt opening the volume.

After opening the volume, files can be written to the volume and read from it, as with any storage volume.

The speed of the file transfer is limited by the speed of the SD-Card interface, which is implemented using SPI as opposed to the parallel SD interface. The MSC API implements a double-buffering feature, essentially multitasking the USB and media sides of the transfer which mitigates this problem. See the M4 example for details.

3.5.2.2 Implementation Comments

In the Descriptor Tool, certain information was set up to describe this MSC interface.
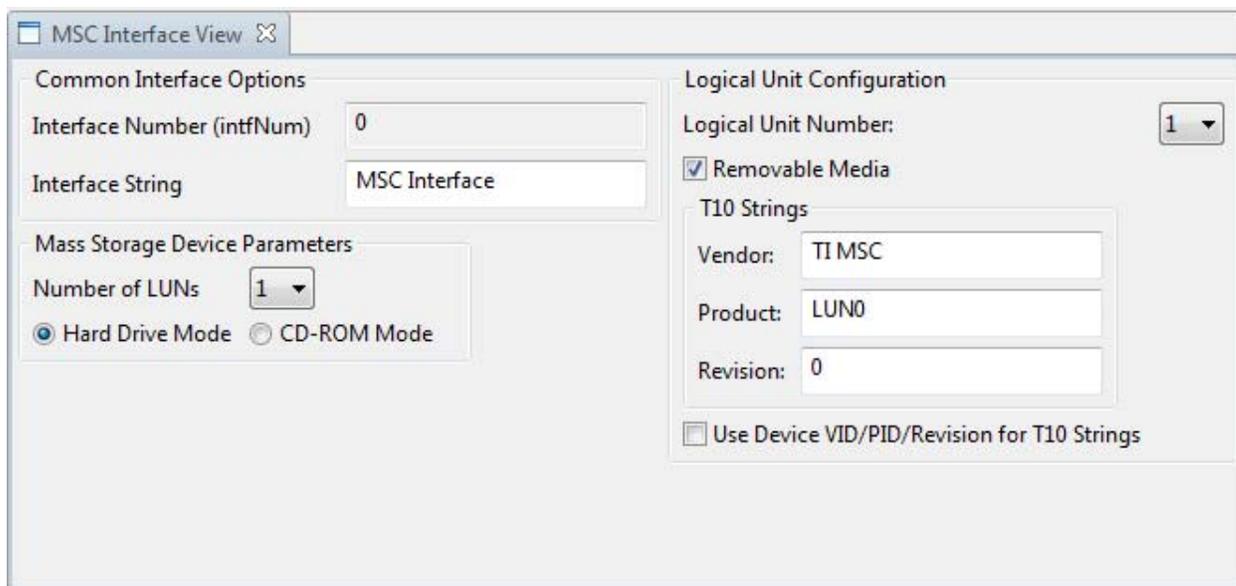

Figure 33.   Defining the MSC Interface, in the Descriptor Tool

The device is selected to have only a single *logical unit (LUN).* The device will operate as a hard drive, rather than a CD-ROM. (Most devices fall in this category; the Descriptor Tool's help pane text explains this.) Finally, the "Removable Media" box is checked, because SD-cards can always be removed at any time by the user.

In the code, a static buffer `RWBuf` is allotted for transferring data between the application and host. This will be used every time the API requests the application to "process" a buffer. `USBMSC_registerBufInfo()` is used to register this buffer with the API.

A pointer to a structure `USBMSC_RWbuf_Info`, named `*RWbuf_info`, is created. This structure is used as an exchange for information related to buffer requests. The application will later poll the *operation* field of this structure to determine if a buffer has been requested. Unlike the buffer, this structure instance is allocated within the API, and the pointer is passed back to the application using `USBMSC_fetchInfoStruct()`. The reason the buffer is allocated in the application is because it's relatively large, and this allows the application to dynamically de-allocate it when it's not needed for USB.

One of the first thing the code does is inform the API of the initial state of the media. It uses FatFs calls to learn this state, and then calls `USBMSC_updateMediaInfo()` to inform the API.

Detection of the SD-Card is usually done with an I/O. The card has a pullup on one of the interface lines, and this pullup can be detected with an I/O pin. The F5529 Experimenter's Board doesn't have this I/O connection, so this method wasn't possible. Instead, a new function `detectCard()` was added to FatFs, which employs a software algorithm to determine if the card is present or not. This call is then called from the application once every second.

Therefore, early in the application, the timer is configured to generate an interrupt once every second. This is usually fast enough to detect the user inserting/removing the card.

Within the main loop's `ST_ENUM_ACTIVE` branch, `USBMSC_poll()` is called. If no READ/WRITE operation is in work, the CPU goes into LPM0. If an operation is active, however, the return value will keep the main loop awake. it will check the *operation* field of `RWbuf_info`, and find a value indicating whether it's a read or write. This marks the beginning of the "buffer operation". The application performs the buffer operation by making appropriate calls to FatFs, to access the SD-Card.

Return values from these FatFs calls are then used to assign a status code to return to the API. The application then calls `USBMSC_bufferProcessed()` to inform the API that the buffer has been fully processed, including the return code. This marks the end of the buffer operation. Based on the return code, the API will handle any necessary interaction with the host.

Note that the application has no awareness of specific SCSI operations. All it is aware of is that the API occasionally asks it to access the media. Through this, it is somewhat aware that a READ or WRITE command has been issued from the host. But these commands may generate mutiple buffer operations, and the application doesn't need to know "where" within that READ/WRITE command the buffer request lies. It simply processes buffers.

3.5.3   Example #M3_MultipleLUN

This example demonstrates the implemention of two logical units (LUNs). It causes two volumes to mount on the host. It is essentially the combination of #M1 (file system emulation) and #M2 (SD-card).

Like #M2, this example requires hardware with an SD-card interface, and is specifically designed to run on the F5529 Experimenter's Board, available from TI's eStore.

Because of the example's similarity to #M1 and #M2, please reference the sections above for those examples regarding its usage.

3.5.3.1   Running It

Simply plug into the USB host.   It should enumerate as a storage device, with *two* storage volumes mounted on the system. On Windows, this means two volumes will appear in "My Computer".

Again, please reference the sections for examples #M1 and #M2.

3.5.3.2   Implementation Comments

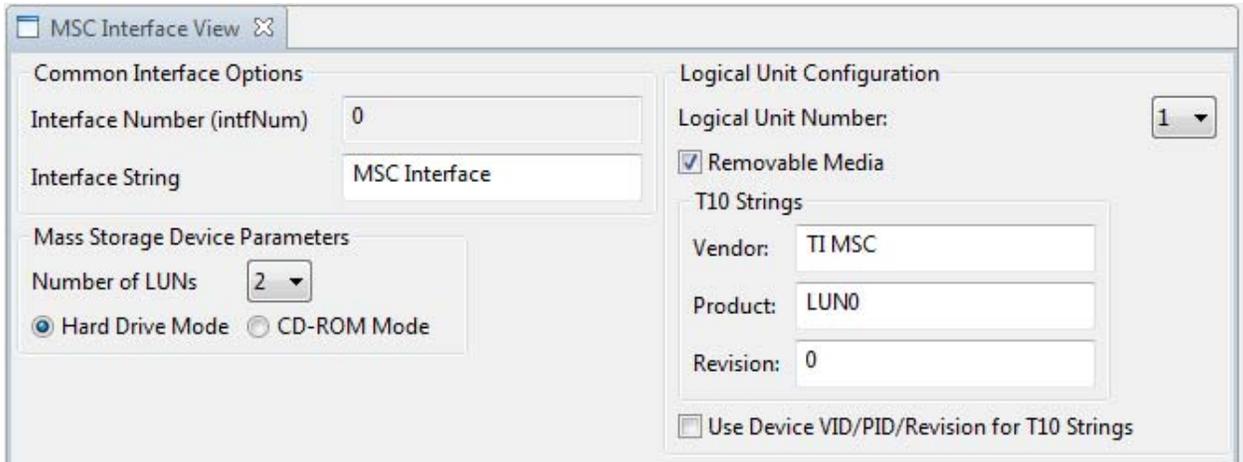Again, the Descriptor Tool is used to set up the interface. Two LUNs are defined. Each LUN has its own settings.

Figure 34. Two-LUN Device; Viewing the First LUN's Data

The area on the far right is specific to each LUN. The pulldown menu in the far right corner selects which LUN's data is shown.
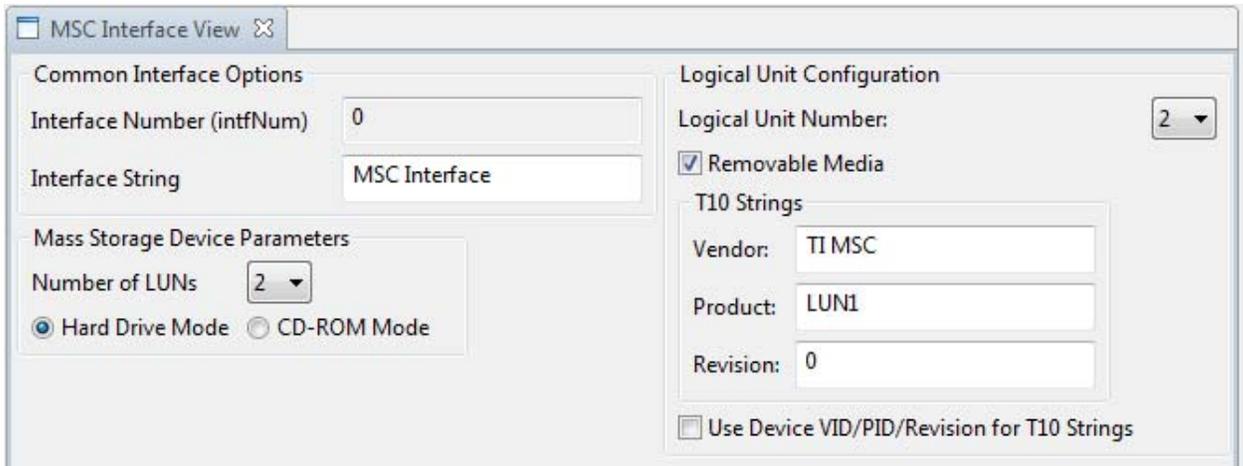

Figure 35. Two-LUN Device; Viewing the Secon LUN's Data

Both LUNs are defined as having removable media. The reason for the file-system-emulation volume requiring this is unique to the volume this example chose to emulate; the example doesn't work properly if configured as non-removeable. The reason the SD-card volume is marked as removable is because the media in fact is removable.

In most applications, the T10 strings have no effect and can be ignored. For anyone wishing to set them, the ability is there.

Unlike the other examples that only have one LUN, this application's handling of buffer operations checks the *lun* field of the `USBMSC_RWBuf_Info` structure. Obviously, the file system emulation volume (which uses no file system software) is handled much differently than the SD-card (which is accessed through the FatFs file system software).

Since there can be only one MSC interface in a device implemented by this API, it is assured that only one SCSI command can be received from the host at a time, and thus only one buffer operation can be active at a time. This simplifies the application's handling of buffer operations.

3.5.4   Example #M4_DoubleBuffering

This example is the same as the #M2 example but uses double buffering. The application specifies two buffers when it calls `USBMSC_registerBufInfo().` Both buffers need to be the same size. The double buffering improves MSC performance.

As with #M2 Example, this example requires hardware with an SD-card interface, like the F5529 Experimenter's Board, available from TI's eStore.

3.5.4.1 Running it

Simply plug into the USB host.   It should enumerate as a storage device, and the storage volume should be mounted on the system. On Windows, this means the volume will appear in "My Computer". If a card is present in the hardware, then the volume can be opened. If no card is present, then attempting to open it will probably result in a warning that no media is present. Insert one, and then re-attempt opening the volume.

After opening the volume, files can be written to the volume and read from it, as with any storage volume.

The speed of the file transfer is limited by the speed of the SD-Card interface, which is implemented using SPI as opposed to the parallel SD interface.

3.5.4.2 Implementation Comments

Please read the Implementation Comments for the #M2 Example, as they apply to the #M4 example as well.

In the #M4 example the USB API tries to keep both the host and the SD-Card busy. When the host initiates a READ operation the API will ask the application to fill the two available buffers one after the other by setting fields in `RWBuf`.. When the first buffer is filled by the application, the API will immediately start a transfer to the host. The API keeps track of whether application is done processing buffer to make a full buffer available to the host as soon as possible.

When the host initiates a WRITE operation, the API will make available both buffers for the host to fill. When the first buffer is filled by the host, the API will set fields in `RWBuf` to indicate a WRITE operation. The API keeps track of whether application is done processing buffer to make an empty buffer available to the host as soon as possible.

In the ideal case both interfaces will be busy. However we do see that in most cases we are limited by the speed of the SPI interface.

3.5.5 Example #M5_CDROM

3.5.5.1 Running It

This application shows how to emulate a CD-ROM . What makes this useful is that an ISO9660 CD-ROM image is the only way to autorun a file on all three major host operating systems.

This example uses an SD-card as the underlying storage medium, although it could be adapted to other media. Because of the SD-card requirement, this example requires hardware with an SD-card socket, like the F5529 Experimenter's Board.

Before running #M5, use the #M2 example to create the ISO9660 image on the emulated CD-ROM device. Running #M2 causes a normal removable mass storage device to be reported to the host. From the host operating system, format the volume as a generic FAT volume. Then, onto the root directory of this volume, copy an existing ISO9660 CDFS image file from a CD-ROM. It should be named *mount.iso*.

Then, download the #M5 example to the target MSP430 device. Once downloaded and plugged into the host, the application will show up as a generic CD-ROM device, with the designated *mount.iso* file.

If any autorun information is present in the ISO file, the host will take the appropriate actions to do so.

If the *mount.iso* file is not found by the application, an empty CD-ROM device will be reported to the host.

3.5.5.2 Implementation Comments

Placing the ISO image on a FAT volume is an easy way to create the target media/volume, which is why example #M2 and an SD-card was borrowed for this purpose.

The host needs to be told this is a CD-ROM drive. This is done with the Descriptor Tool.
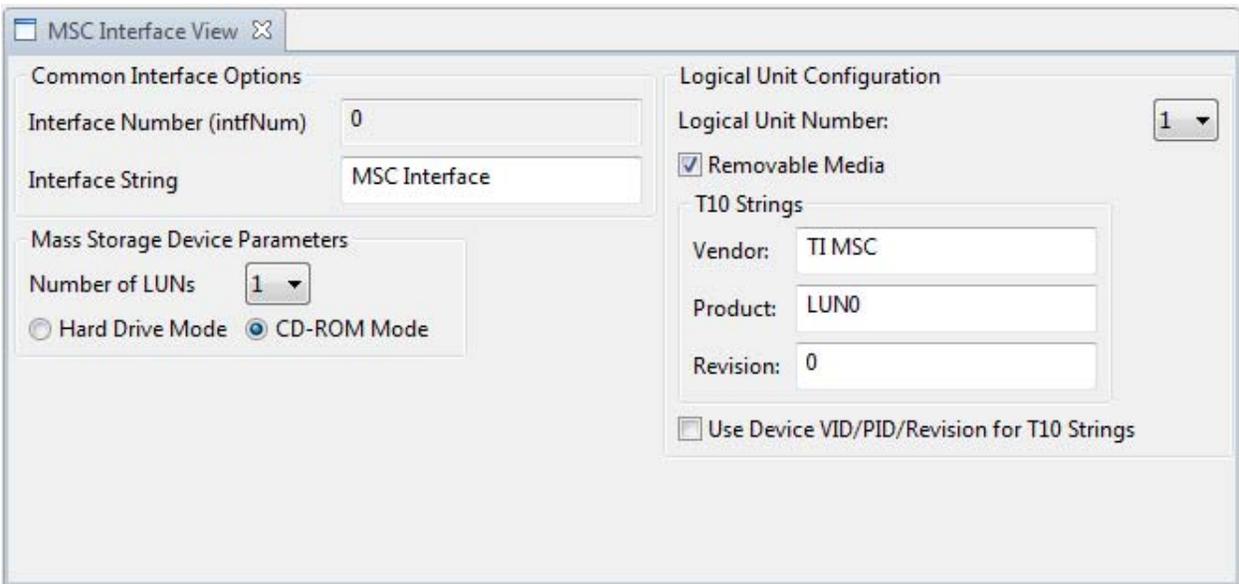
Figure 36. Configuring as a CD-ROM Drive in the Descriptor Tool

Before the main loop is executed, a call to the FatFs library is made, to open the FAT volume contained on *LUN 0*. Once the volume is open, a call is made to the FatFs API to open the *mount.iso* file, contained in the root of the file system. If the file does not exist, the CD-ROM drive is reported as empty to the host.

With the ISO file open, whenever a READ command is received from the host, a FatFs read call is performed on the *mount.iso* file. LBA addresses and offsets in the ISO file are calculated and generated on-the-fly. WRITE operations are not supported, since CD-ROM devices are write-protected.

3.6 Composite Examples

3.6.1 Example #CH1_term2hidDemo

3.6.1.1 Running It

This application shows a composite device consisting of a CDC and HID-Datapipe interface. A terminal application is used to communicate with the CDC interface, and the HID Demo App is used to communicate with the HID interface. The MSP430 application echoes data between the interfaces.

Since this example contains both a CDC and HID-Datapipe example, refer to both Sec. 3.1.1 and Sec. 3.1.2 for general information on running this example.

It's important that the correct INF file be used, because the PID for this example is different than the PIDs for single-CDC or single-HID interfaces. As with all the examples, an INF file is provided in the same directory. Be sure to use this INF for this example.

When the installation is complete, open connections with both a terminal application, and the HID Demo App. For the HID Demo App, set the VID to 0x2047 and the PID to 0x0302.

Typing characters into the terminal application will cause them to appear in the HID Demo App, and vice versa.

3.6.1.2 Implementation Comments

The active-enumerated portion of the main loop consists mostly of two blocks which evaluate flags set by `USBHID_handleDataReceived()` and `USBCDC_handleDataReceived()`, respectively. When any data is received, it's echoed onto the other interface.

CDC and HID are handled somewhat differently because of differences in the way that Hyperterminal and the HID Demo App send data. The HID Demo App sends data in one action when the 'send' button is pressed (whether or not it includes return characters). Hyperterminal sends data as it's typed, and the string is return-delimited. If a terminal application is used that handles this in the same way as the HID Demo App, different behavior will be seen. This could be resolved by altering the CDC code to look just like the HID code, but inverting the CDC and HID calls/flags. (The CDC calls and the corresponding HID-Datapipe calls function almost identically, allowing easy migration between the two.)

3.6.2 Example #CC1_term2term

3.6.2.1 Running It

This application is just like the preceding one, except it shows a composite CDC+CDC device. Two instances of a terminal application are opened, each opening a COM port associated with a different CDC interface in this device. The MSP430 application echos data between the two interfaces.

Since both interfaces in this example are CDC, refer to Sec. 3.1.1 for general information on running this example.

Run the example and enumerate the device. Unlike the examples containing one CDC interface, this one should result in *two* device installation procedures. Once again, it's important to use the INF file provided in this example's directory. This is because the PID is once again different than the other examples, and equally important is that the USB interface set has changed.

Once the device installation procedures are finished, two new COM ports should be present on the system. Run Hyperterminal or another terminal application, and open one of these COM ports. Run a second instance of this application, and open the other new COM port.

Once both ports are opened, sending characters from one instance of the terminal application (and pressing return) will cause them to appear in the other, and vice versa.

3.6.2.2 Implementation Comments

This example is written almost exactly like #CH1, except using CDC for both interfaces. The code for each interface is exactly the same, except for using intfNum values of 0 versus 1.

3.6.3 Example #HH1+_hidDemo2hidDemo

3.6.3.1 Running It

This application is just like the preceding ones, except it shows a composite HID+HID device.

Since both interfaces in this example are HID-Datapipe, refer to Sec. 3.1.2 for general information on running this example.

Run two instances of the HID Demo App. For both of them, set the VID to 0x2047, and the PID to 0x0314.

After pressing "Set VID/PID" in each instance's window, a value should appear in the "Serial Number" menu. Both instances should show "HID0" and "HID1" in the interface list. In one of the instance's window, select interface "HID1", leaving the other as "HID0". Then press the "Connect" button in each instance window, to initiate connections with the HID interfaces.

Now, type text into one window and press "Send"; the data appears in the other window, being echoed by the MSP430 from one interface to the other.

3.6.3.2 Implementation Comments

Once again, this example is written almost exactly like #CH1, except using HID for both interfaces. The code for each interface is almost exactly the same.

3.6.4 Example #CHM1_term2HidDemo_2LUN

This example is essentially a combination of the #CH1 and #M3 examples. It combines interfaces from all three device classes (CDC/HID/MSC), while making the MSC multiple-LUN.

Since this example contains a CDC interface, and HID-Datapipe interface, and an MSC interface, refer to Sec. 3.1.1, Sec. 3.1.2, and Sec. 3.1.4 for general information on running this example.

Reference the commentary for #CH1 and #M3 for more information.

3.7 SYSBIOS Examples

These examples show how to use the USB API with TI's SYSBIOS real-time operating system (RTOS).

3.7.1 Example #CHM2_tasks

This example shows the use of SYS/BIOS RTOS with the USB API. It has one CDC, one HID, and an MSC interface

This application has three SYS/BIOS tasks:

- cdc0Task: manages the CDC interface, and implements a simple echo
- hid0Task: manages the HID interface, and implements a simple echo
- lun0Task: manages a File System Emulation drive (see Example #M1)

Each of these tasks start up, and pend, on semaphores. When data is received on any of the USB interfaces, the associated task is made ready to run by a `Semaphore_post()` call from within the USB event handlers.

This example also uses an idle background function `myIdleFxn()` to check the USB connection state. It enters a low power mode when needed. The `USBMSC_poll()` function, a normla part of handling an MSC interface, is also called within this idle thread.

This application can easily be modified to have multiple CDC and HID interfaces or an MSC with multiple LUNs.

3.7.2 Example #CHM3_SWIs

This example shows the use of SYS/BIOS RTOS with the USB API. It has one CDC, one HID, and an MSC interface

This application has three SYSBIOS "swi's" (software interrupts):
- cdc0Swi: manages the CDC interface and implements a simple echo
- hid0TSwi: manages the HID interface and implements a simple echo
- lun0Swi: manages a File System Emulation drive (see Example #M1)

When data is received on any of the USB interfaces, the associated Swi is made ready to run by a `Swi_post()` call from within the USB event handlers.

This example also uses an idle background function `myIdleFxn()` to check the USB connection state. It enters a low power mode when needed. The `USBMSC_poll()` function, a normla part of handling an MSC interface, is also called within this idle thread.

This application can easily be modified to have multiple CDC and HID interfaces or an MSC with multiple LUNs.

3.8   General Examples

3.8.1 Example #G1_reduceInterruptLatency

This example demonstrates how to eliminate the interrupt latency associated with stabilization of the XT2 oscilllator the the locking of the USB PLL.

During USB execution, the USB interrupts all have very short latency except the ones for USB resume and a VBUS-on event (the attachment to a host via the USB cable). These can take a few milliseconds, because they activate the XT2 oscillator and PLL. This happens inside an interrupt context, so while they are happening, your application won't be able to service other interrupts.

A new feature introduced the v4.0 and later of the API provide a way to break the USB resume process into three pieces:
1. starting the XT2 oscillator stabilization (happens interally)
2. starting the PLL (begun by a call to the API function `USB_enable_PLL()`)
3. finishing the process (begun by a call to the API function `USB_enable_final()`)

USB event handlers are triggered by the completion of #1 and #2. These give control back to the application, and the application is responsible for ensuring XT2 stabilizes and the PLL locks. It may do this using delay period implemented with a timer. During this timer delay, the application can remain in a normal context (rather than interrupt context), servicing any interrupts that arise.

With this feature in place, all USB interrupts finish very quickly (see the Release Notes for parametrics).

With the above in mind, this example demonstrates this feature. Its token USB functionality is to send "Hello World" repeatedly, but the relevant portions of the example are in the timer ISR and the event handlers `USB_handleCrystalStartedEvent()` and `USB_handlePLLStartedEvent()`.

Note that to use this feature, you must define `USE_TIMER_FOR_RESUME` in the project settings. This internal #define causes the API to break the resume event into the three phases above. After defining `USE_TIMER_FOR_RESUME`, call the special functions shown in the timer ISR.

The sequence of events in this example is:
1) USB resume event occurs; some initial internal USB API code executes
2) `USB_handleCrystalStartedEvent()` event occurs; calls application function `startTimerForOscStabilize()`
3) `startTimerForOscStabilize()` starts the timer
4) Timer expires; ensures <u>osc</u> stabilization, then calls `USB_enable_PLL()`
5) `USB_handlePLLStartedEvent()` event occurs, calls application function `startTimerforPLLlock()`
6) `startTimerforPLLlock()` starts timer
7) Timer expires, calls `USB_enable_final()`

The USB PLL is now stable, and USB functionality resumes as normal. .

Appendix A. Launching the Device Manager on Windows

Windows, Linux, and the MacOS all have a means of viewing the devices attached to them. This is very useful when evaluating or developing USB devices.

This Examples Guide uses Windows as an example, and thus it's described here how to launch the Windows Device Manager.

To open the Device Manager, navigate to the Start Menu, select "Run…", enter "devmgmt.msc", and press "Enter" or "OK".
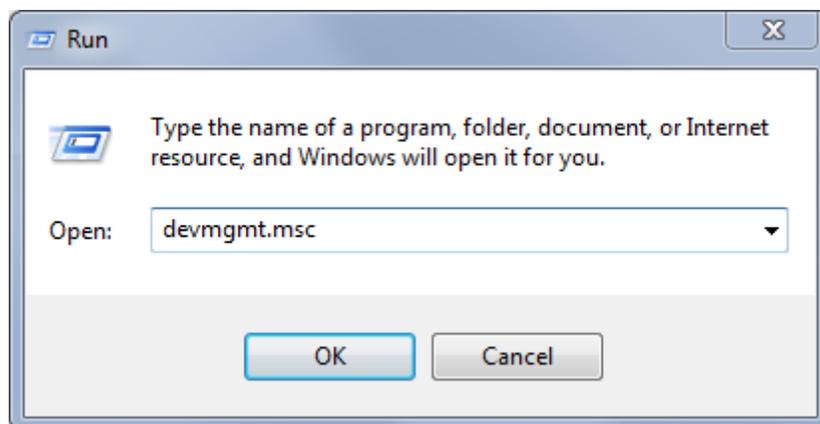

Figure 37.   Starting the Windows Device Manager

If Windows asks if you want to allow the program to make changes to your computer, select "Yes"; however, the Device Manager is very useful for viewing purposes, without having to change anything.
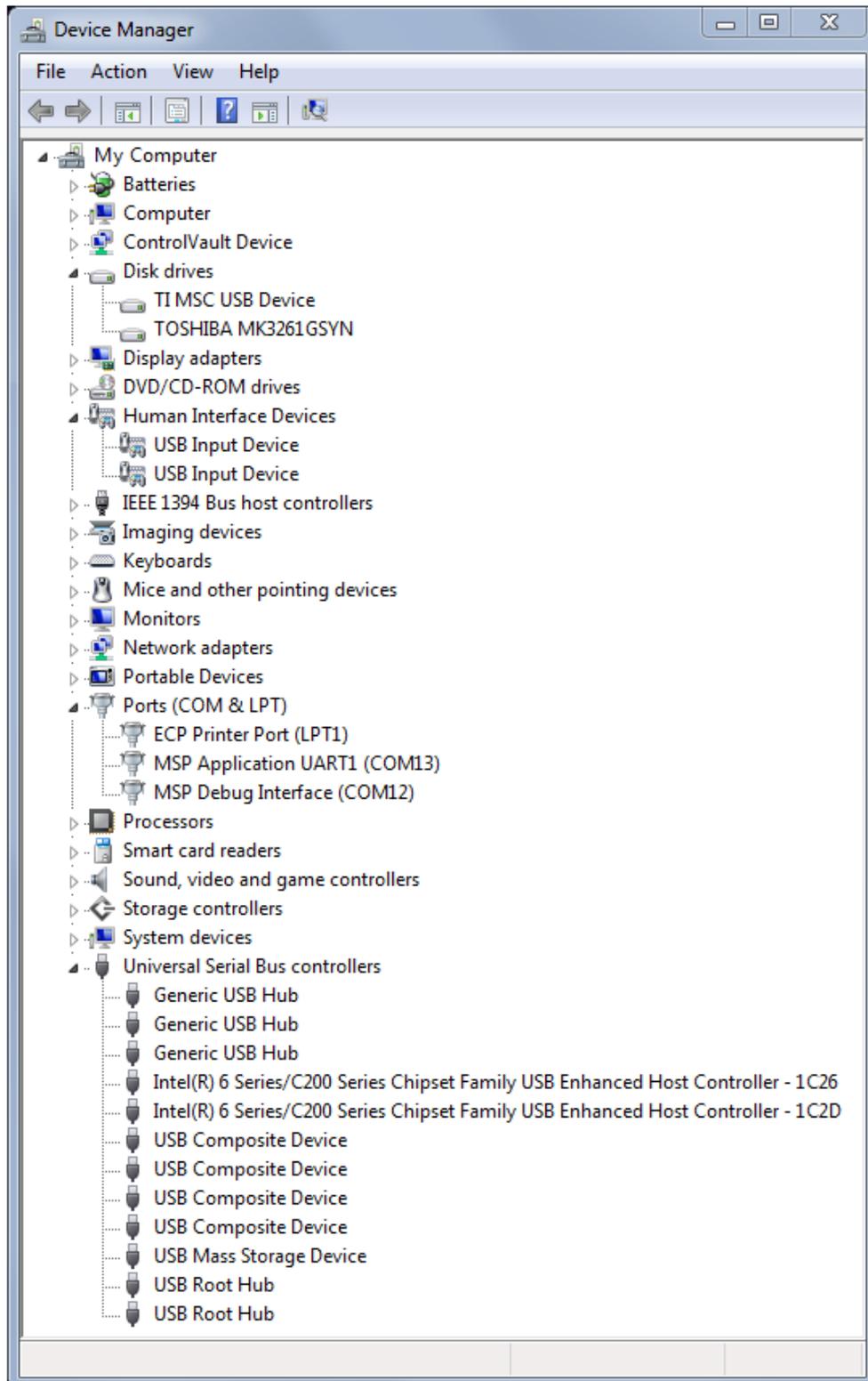
Figure 38.   The Windows Device Manager

Groups in the DevMan that are relevant to MSP430 USB work include:

- Ports (for CDC interfaces, resulting in virtual COM ports)
- Human Interface Devices (for HID interfaces).
- Disk Drives (for any drives that have been mounted via an MSC interface)
- Universal Serial Bus controllers. (Hubs and MSC interfaces appear here, as do root entries for composite USB devices.)

The Device Manager is also very useful during debug of a USB application. The USB API Programmer's Guide, also inside this USB Developers Package, contains a section on how to debug your USB application; see this document for more information on using the Device Manager.