TEXAS INSTRUMENTS

March 2011

Programmer's Guide: MSP430 USB API Stack for CDC/HID/MSC

MSP430

ABSTRACT

The MSP430 USB API implements three USB device classes: the Communications Device Class (CDC), the Human Interface Device (HID) class, and.the Mass Storage class (MSC). It is designed for easy creation of USB devices on the MSP430.

Contents

1 Introduction

The USB API (application programming interface) stack for the MSP430 is a turnkey API. It makes it easy to implement a simple USB data connection between an MSP430 and a USB host. It includes support for three common USB device classes:

- Communications Device Class (CDC)
- Human Interface Device class (HID)
- Mass Storage Class (MSC)

These APIs are designed to minimize the USB knowledge required to write an application:

- All USB protocol is handled automatically by the API
- The data interface port seen by the application is very simple to use
- USB descriptors and stack configuration are automatically handled by the *MSP430 USB Descriptor Tool*

The user shouldn't need to modify the API source. However, for experienced USB programmers, the source is made open and available for editing. Accessing the API's source can also be useful for system debug and gaining a deeper knowledge of the USB system.

Application examples are included with the API.

2 The MSP430 USB Developers' Package

This API is part of a suite of tools TI provides to make USB easy on the MSP430, including:

- *MSP430 USB Descriptor Tool:* a code generation tool that automatically generates reliable descriptors for use with this API, for any combination of USB interfaces. It saves the developer's time and reduces the chance for errors.
- *MSP430 USB API Stacks:* an API for implementing common USB device classes (CDC/HID/MSC)
- *Windows HID API:* a Windows API that complements the MSP430 HID API stack, simplifying creation of a general-purpose USB device using HID
- *MSP430 USB Field Firmware Updater:* a Windows Visual Studio project for an application that downloads firmware to the MSP430 over USB, just by double-clicking. Simply insert the new firmware image file and compile.

3 Glossary

3.1 USB Definitions

- **USB-IF:** The USB Implementers Forum. This is the standards body the defines USB specifications, governs USB certification, runs compliance workshops, and owns the legal rights to the USB logo.
- **USB Host:** USB is hierarchial, with one (and only one) host that controls all communication.
- **USB Device:** Also called a USB "function". This is a logical or physical entity on the bus that contains one or more *USB interfaces*. It possesses one upstream-capable USB connector.
- **USB Hub:** A device that provides communication between one upstream connector and multiple downstream connectors, allowing more USB devices to be attached to a host. In any given bus configuration, a device is either a host, device, or hub.
- **Device Class:** A defined USB protocol for a particular class of devices. Common device classes include the Communications Device Class (CDC), Human Interface Device (HID) class, and Mass Storage Class (MSC).
- **USB Interface:** A logical USB entity that performs a particular function. An interface is typically associated with a particular *device class* – for example, a "CDC interface".
- **Composite USB Device:** a physical *USB device* (one USB connector) that contains more than one *USB interface* – for example, two CDC interfaces, or CDC+HID. The host *enumerates* each interface as a separate logical entity.
- **USB Descriptors:** Data structures contained within a physical USB device that describe the device (including the interfaces it supports) and its capabilities. The host reads these during *enumeration.*
- **Enumeration**: The process by which a host interrogates a physical USB device to determine what it is, and loads an appropriate driver so that the host application can interface with it. Enumeration happens every time the device is attached.
- **Device Installation:** The first time a USB device is enumerated, the host may perform one-time functions to "install" the device. For example, Windows records information about the device in the system registry, using the device's VID/PID as an index. In subsequent enumerations, the host draws from the registry for much of its information about the device. Device installation may be "silent" (mostly invisible to the end user), or in the case of CDC on Windows, may require user action.
- **INF (*.inf) file:** A text-based file required during any USB device installation on Windows, allowing Windows to associate the device with a particular driver. For some device classes, Windows contains the INF internally, allowing for a silent device installation. For CDC, Windows prompts the end user for the INF file.
- **Vendor ID (VID)**: A unique 16-bit value assigned by the USB Implementers Forum to a particular USB hardware vendor.

- **Product ID (PID)**: A unique 16-bit value assigned by a USB hardware vendor to one of its products. A VID/PID pair uniquely identifies a product type. (As a rule of thumb, if the USB descriptors of two products differ in any way, they should have different PIDs as well.)
- **USB Serial Number:** A unique string that allows a host to differentiate between devices attached to it that contain the same VID/PID values.
- **VBUS:** The host is required to make 5V power available to the device via the USB cable. The name of this power rail is *VBUS*. In addition to sourcing power, the USB device uses VBUS to determine whether an active host is present. Devices often respond to a VBUS-on event by asserting their presence to the host, by pulling up the D+ signal.
- **Pipe:** A single line of communication between host and device. Pipes are either IN (into the host) or OUT (out of the host). They are characterized by a particular *transfer type* (i.e., bulk or interrupt).
- **Endpoint:** The end of a *pipe*. It acts as a "mailbox" on the USB device for that pipe. A device usually has more than one active endpoint. When the host communicates on the bus, it first identifies the physical USB device, then the endpoint number within that device that it wishes to speak to. Endpoints are assigned specific functions according to the *USB interfaces* that were created. HID/MSC each use one IN and one OUT endpoint, while CDC uses two IN and one OUT endpoint. In the MSP430 API stacks, endpoint management is fully automated by the Descriptor Tool.
- **Bulk Transfers:** One of four data transfer types on the USB bus. Bulk transfers are designed for moving high volumes of data. They're capable of using any free bandwidth on the bus (that is, bandwidth not already used by the other transfer types). This allows them to achieve the highest data rates; but they're given no reserved bandwidth, so on a busy bus, bulk transfers might receive small bandwidth or experience high latency. Transfer types are fully determined by interface selection. For example, CDC and MSC interfaces use bulk transfers.
- **Interrupt Transfers:** Another of the four USB data transfer types. Interrupt transfers are designed for guaranteed latency and bandwidth. However, the bandwidth is limited to only a single USB packet (64 bytes for full-speed USB) per frame (1ms). This leads to a maximum bandwidth of 64KB/sec. Transfer types are fully determined by interface selection. For example, HID interfaces use interrupt transfers.
- **USB speeds:** A USB connection is characterized by one of four speeds: low-speed (1.5Mbps), full-speed (12Mbps), high-speed (480Mbps), or super-speed (4.8Gbps). MSP430 is a USB 3.0 full-speed device.

3.2 API Stack Definitions

- **API Stack:** The name assigned to the USB API that is provided by TI for MSP430.
- **API Space:** The actual API source files. These files are generally meant to be left unedited, although advanced users are free to change it, as needed.
- **Application Space:** A functioning stack based on this API requires an application that makes calls to it. This application exists in the *application space*. It is owned by the software

developer using the API to develop an application. Technically speaking, the application space includes the event handlers, since these must be written by the software developer.

- **Events.** The API keeps the handling of USB interrupts internal. Instead, the API generates *events*. The software engineer can write *handlers* for events. Events function similarly to callbacks, with the main difference being that event handler names/parameter lists are pre-defined.

- **HID-Datapipe Interface:** HID interfaces in this API can either be *datapipe* or *traditional.* Datapipe eases the creation of general-purpose applications over HID. A defining feature is its usage of the API's default report descriptor, which defines a report with a "size/data" field format. The distinction between datapipe and traditional is made only by which function calls the application uses to access the HID interface.

- **HID-Traditional Interface:** This is a traditional HID interface. A defining feature, compared with datapipe, is a report descriptor that is customized by the software engineer, instead of the default one generated by the Descriptor Tool.

- **User Buffer:** All data transfer with the API stack involves a user buffer as a point of mutual exchange. It is defined by *address* and *size* parameters. It can be any contiguous block in the MSP430 memory map, and it can be of any size.

- **USB Endpoint Buffer:** The actual USB endpoint, limited in size to 64 bytes. The application does not access it directly, but rather the API stack exchanges data between the endpoint buffer and the *user buffer.* It automatically packetizes/de-packetizes data in the background, as the buffer is filled/emptied by the host.

- **Send/Receive Operation:** When using a CDC interface or HID-Datapipe interface to move data, all sending/receiving takes place in the context of an *operation* of a specific size. Operations may be handled in the background while the application performs other tasks. The operation is complete when all data has been moved between the endpoint buffer and user buffer.

4 System Overview and Architecture

This section describes the overall USB system, the USB API stacks for the MSP430, and details on the CDC/HID interfaces.

4.1 MSP430 USB API Stacks: Overview

4.1.1 Devices Classes Supported by the MSP430 API Stacks

USB *device classes* define a USB protocol to support a given type of device. MSP430 provides API stacks for three common USB device classes:

- Communications Device Class (CDC)
- Human Interface Device class (HID)
- Mass Storage Class (MSC)

For a full discussion of the tradeoffs between these, see the application note *Starting a USB Design with MSP430 MCUs.*

These three API stacks are distributed together as a single project, sharing a common USB layer.

4.1.2 Development Environments Supported by the API

The USB API stacks compile and run on both the IAR and CCS environments for MSP430, including the size-limited free versions available for download from TI. (See http://www.ti.com/msp430 to obtain these.) It may also run with other environments for MSP430, but it has not been tested on them.

Note that MSC does not run on the size-limited IAR Kickstart, which is limited to 8K of object code. Instead, use the free version of CCS (limited to 16K), or licensed versions of either environment.

4.1.3 Hardware Requirements

The same API stack distribution runs on all three USB-equipped families from MSP430:

- F552x/551x
- F550x (including the F5510)
- F663x/563x

(Note that the F5510 is a member of the F550x family, and not the F551x family.)

The API stack is designed to automatically adapt to whatever device derivative is chosen in the IAR/CCS project settings. The only required action is to select the proper device. The code within the API remains unchanged no matter which device is selected. However, at the application level, the examples distributed with the API need to change certain aspects of operation depending on which device is chosen, in the areas of clocks and port register initialization. In these areas of the code, compiler directives are evaluated, whose settings are based on the device selected in the project settings.

Note that the application examples initialize the port I/O registers. The exact set of port registers tends to vary by device, so the examples configure themselves for the largest device within each family. If compilation fails because of individual lines of code that configure the ports, the offending lines can be commented out without affecting USB operation. If power is being measured, however, it's important to ensure that the port I/O registers are configured to completely eliminate any floating input pins, as these can increase current consumption.

4.1.4 USB Certification

The API stacks have passed USB certification testing for all three device classes (CDC/HID/MSC).

4.1.5 Stack Organization

A software stack using this API is shown in the figure below.

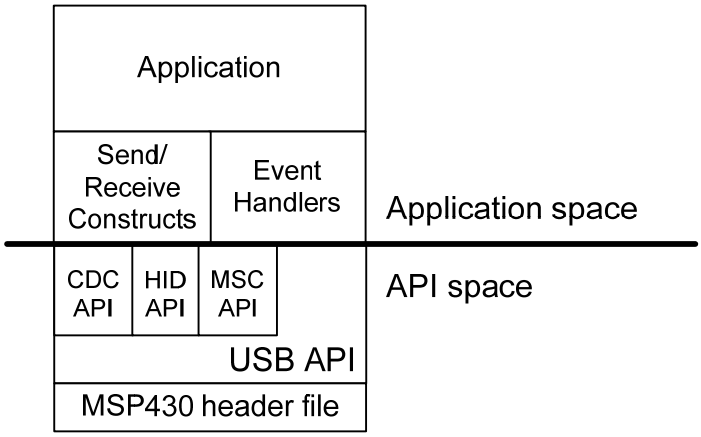


Figure 1. MSP430 API Stack Diagram

All three API stacks (CDC/HID/MSC) share a common USB layer. The application makes calls to these APIs, and also directly to the USB layer.

The stack is divided into *API space* and *application space.* In most cases TI recommends to only modify application space, not the API itself. This helps preserve USB compliance, increasing the chance of passing USB certification and avoiding complications. The send/receive constructions and event handlers are constructs that are provided with the API, but they are considered to be in application space.

The files are as shown in the table below.

Table 1. MSP430 USB API Stack Files

	User-Modify?	Filename	Description
Application space	Yes	*main.c/h*	User application
		usbConstructs.c/h	Contains example constructs for send/receive operations. The functions here reflect the approaches described in Sec. 12
		usbEventHandling.c/h	Event-handling placeholder functions.
USB API space (config)	Generate from the Descriptor Tool	*descriptors.c/h*	*Descriptors.c* contains data structures that define the USB descriptors the device will report to the host. A default descriptor set is provided, which can be customized with the *MSP430 USB Descriptor Tool*. *Descriptors.h*contains the configuration constants and additional descriptor information, as described in Appendix A.
		UsbIsr.c	USB interrupt service routine handler, and related functionality
USB API space (core)	No	*Usbcdc.c/h*	CDC-related functionality
		UsbHid.c/h *UsbHidReportHandler.c/h* *UsbHidReq.c/h*	HID-related functionality
		UsbMscScsi.c/h *UsbMscStateMachine.c/h* *UsbMscReq.c/h*	MSC-related functionality
		usb.c/h *UsbIsr.h*	Functionality common to all USB applications.
		dma.c/h	Functions related to DMA transfers
F5xx/6xx Core Library	No	*hal_pmm.c/h* *hal_ucs.c/h* *hal_tlv.c/h*	MSP430's standard library for the F5xx architecture. The USB API uses it to handle the PMM (power) and UCS (clocking) modules, and for reading the TLV structure (Tag-Length-Value) in MSP430 flash to obtain the device's unique die ID number. This library is available outside this API from http://www.ti.com/msp430, as literature number *slaa448*.
Header files	No	*device.h*	Controls the device derivative for which the stack is configured. In IAR/CCS, it receives its direction from the IDE's project settings.
		defMSP430USB.h	Definitions related to the MSP430 USB module
		types.h	Data type definitions
		msp430fxxxx.h	Standard header file for the MSP430 device derivative being used. This is included in the software development environment, outside this API. Selection is automatically controlled by *device.h.*

The directory structure for the API is shown in the figure below.

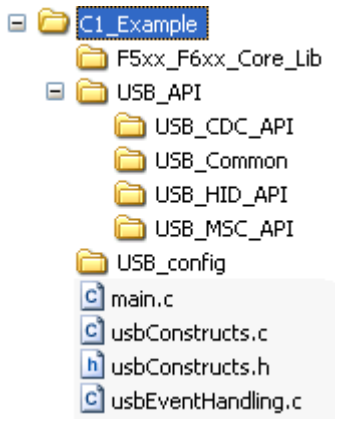


Figure 2. Directory Structure

The directories are roughly similar to the categories in Table 1.

USB_config is where output from the Descriptor Tool should be placed (see Sec. 5). Files in the root directory are considered to be application space, and thus are editable – keeping in mind that the structure of the event handler functions should not be edited.

4.1.6 Usage of MCU Peripheral Resources

Within the API, two resources are used without restriction -- the USB module (with its associated physical device pins), and one DMA channel. The application should avoid directly writing to either the USB module or to the DMA registers that control the selected channel.

The pins associated with the USB module can be used as special I/O pins when not used for USB. The application has permission to manipulate these pins directly only when the USB module has been disabled using the *USB_disable()* call. Once the USB module is enabled using *USB_enable(),* only the API has the right to control these pins.

A DMA channel can be assigned to the API. This is controlled with the Descriptor Tool. All USB interfaces share the same DMA channel, for both transmit and receive.

These are the only peripherals used by the API.

4.1.7 Memory Requirements

Code and data memory requirements are shown below, for various configurations.

Table 2. Memory Usage

Interface Configuration	Code (flash)	Data (RAM)
CDC	5.1K	268 bytes
HID	5.2K	260 bytes
MSC	8.3K	698 bytes
CDC+CDC	5.4K	294 bytes
HID+HID	5.4K	284 bytes
CDC+HID	6.9K	293 bytes

The estimates are from the IAR environment, with the compiler set with the “high” optimization setting (with priority on ‘balanced’). Similar results are obtainable with CCS.

These estimates also assume a “minimal” application – one that only consists of calling every USB function, with no actual application functionality. No RAM user buffers are allocated. Although not practical, this is a baseline benchmark that can be cleanly applied to any application, with the numbers reflecting only the API itself.

Because of the shared USB layer, adding more interfaces only has incremental effects on memory requirements.

The API’s RAM usage is static; no heap space is required.

In IAR, the environment gives a choice of using the DLIB standard library, or the legacy CLIB. To compile the API stacks, DLIB must be selected.

4.1.8 Using an RTOS

The MSP430 USB API stacks were designed to not require an RTOS. However, the API stacks are intended to be straightforward to port to an RTOS.

4.1.9 Support for Composite USB Devices

A *composite* USB device is a single physical device containing multiple USB interfaces. A USB interface is defined by the USB device class it supports – for example, CDC, HID, or MSC. Thus, given these three classes, a USB device might be:

- a CDC device only
- an HID device only
- an MSC device only
- CDC+HID in composite
- HID+MSC in composite
- CDC+CDC in composite
- HID+HID+HID+HID+MSC in composite

and so on.

A software engineer may wish to create composite devices for a variety of reasons. If two COM ports are desired, this would be accomplished with two CDC interfaces in composite. MSC doesn't lend itself well to generic command/status information, so creating a composite device with MSC and either CDC or HID can enable a device that has both storage capability and general communication.

The MSP430 USB API stacks can be very easily configured to create any combination of composite interfaces, using the *MSP430 USB Descriptor Tool*. All that remains is to write the application. (See Sec. 5 for more information about the Descriptor Tool.)

The number of composite interfaces is limited only by the number of endpoints in the MSP430 USB module. As examples of what is possible, MSP430 has enough endpoints to create:

- Three CDC interfaces
- Two CDC interfaces and two HID interfaces
- Two MSC interface, three HID interfaces, and one CDC interface

This is more than enough for most applications.

Note that only one MSC interface can be implemented. If more than one storage volume is desired, use multiple logical units (LUNs).

4.1.10 Release Notes, and Migration from Previous Versions

For those migrating from previous versions of this API, release notes have been included within the software distribution. They include all changes from the previous versions, and instructions for migration.

4.2 The Communications Device Class (CDC) API

4.2.1 CDC Overview

An MSP430 running this API and attached to a USB host via USB will establish a virtual COM port on that host.

Note: The term *COM port* is specific to the Windows group of operating systems, but the MacOS, Linux, and other operating systems provide similar mechanisms. "COM port" will be used in this document to refer to all of them, and the term "PC" may be used to refer to any USB host.

COM ports are a popular, simple software mechanism through which a host can communicate with a peripheral. Originally designed for RS232 serial ports, it's often today used with other protocols, such as USB and Bluetooth. Since the physical RS232 port no longer exists, these COM ports are often called "virtual COM ports".

The Communications Device Class (CDC) is one of the standard USB device class protocols. It is supported natively by most host operating systems. This has great advantages, discussed in Sec. 0 below.

This API supports only a subset of the CDC specification. This is because CDC has a scope that goes far beyond virtual COM ports, encompassing a wide range of telecommunications equipment. This API supports the Abstract Communication Model (ACM) of the PSTN subclass of the CDC. The ACM provides for a control mechanism using common V.250 AT commands. This configuration establishes a fully-functional virtual COM port interface.

This API supports up to three CDC interfaces in composite.

4.2.2 Host Considerations

The CDC API Stack has been tested with Windows; MacOS, and Linux.

4.2.2.1 Advantages of Native Host OS Driver Support

Like HID and MSC, the CDC protocol is supported natively by the major operating systems. This has great advantages:

- Less hassle for the OEM (no need to prepare a kernel-mode driver installation)
- Less hassle for the end user (doesn't have to perform one)
- Problems are less likely to occur; leading to greater stability and lower support costs

4.2.2.2 Microsoft Windows

The CDC class is supported by Windows 2000, XP, Vista, and 7.

Although Windows natively contains the driver *binaries*, it doesn't contain an INF file. (The INF file is needed to help Windows associate the USB device with the CDC driver.) Therefore, the OEM must provide an INF file to the end user, and the end user must walk Windows through a *device installation* process. In this process, Windows is guided to the INF file.

If the device is composite, then the INF file needs to be tailored to the chosen set of USB interfaces. The Descriptor Tool outputs a tailored INF file, so there is no need to create one manually.

In addition to a CDC driver, the host needs an application to interface with the device. Any "terminal" application (that is, one designed for communication with COM ports) can be used to communicate with an MSP430 equipped with the CDC API stack.

Windows XP Service Pack 3, Vista, and 7 all support a USB descriptor type called an *Interface Association Descriptor* (IAD). Because of this support, they're able to support multiple CDC interfaces in a composite USB device. However, earlier versions of Windows do not support the IAD, and therefore they do not support composite CDC devices with properly-formatted descriptors; they only support single-interface CDC devices. The Descriptor Tool will alert the user of this, if attempting to create a multiple-interface CDC device. (Windows XP SP2 and earlier do support a particular descriptor format that allows composite CDC devices to be enumerated; however, the format violates the CDC specification. Microsoft now supports composite CDC using the IAD, and the Descriptor Tool does not support this illegal format.)

The CDC API stack has been tested on Windows XP Service Pack 2, Windows XP Service Pack 3, and Windows Vista.

4.2.2.3 MacOS

The CDC class is supported on any recent version of the MacOS. The CDC API stack has been tested with version 10.5.6.

The MacOS does not support the IAD. It is not possible for a Mac to enumerate a composite device that contains a CDC interface; only single-interface CDC devices are supported.

4.2.2.4 Linux

The CDC class is supported on any recent, common Linux distribution. Enumeration of the CDC API stack has been tested with Ubuntu version 9.04, but data transfer has not been fully tested.

This version of Linux supports the IAD, and therefore can work with composite CDC devices.

4.2.3 Transmission Speeds

Full-speed USB is rated 12Mbps. This is a theoretical maximum, and it includes protocol overhead – so it isn't possible for a practical application to achieve this rate for data payload. Further, USB is a system involving many components. Any of these components is capable of reducing the bandwidth.

The following factors can all affect bandwidth:

- Host application. USB is very host-driven, which means the host initiates all data transfers, whether sending or receiving. If the host application doesn't initiate transfers often enough, data will be slower.
- Bus loading. CDC uses bulk transfers, which have the potential to reach the highest data rates by using any spare bandwidth. However, the tradeoff to this is that bandwidth scarcity will cause slowed transfers.

- Software on the USB device. If the bus is fast (that is, if the factors mentioned above aren't limiting), the device's application will become the bottleneck in the system. This is because it must take time to process data received and prepare data for sending.

As a benchmark, the CDC API stack can achieve 788KB/sec (6.3Mbps) under the following conditions:

- 8MHz CPU master clock (MCLK)
- Send data from the host to the device
- The MSP430 application calls *USBCDC_rejectData()* for any data it receives.

The purpose of the last item is to nearly eliminate the device application from being a factor. If the application rejects the data, it doesn't use any time to move the data. In contrast, any real application must handle the data, and thus will get considerably less than 788KB/sec – probably closer to the 200-500KB/sec range.

If bandwidth is a priority, DMA should be used. (This can be enabled using the Descriptor Tool.)

For an application in which bandwidth is primarily limited by MCU data handling, bandwidth is roughly linear to MCLK. Therefore, running at the maximum MCLK can often get bandwidth closer to the 788KB/sec "ideal".

4.3 The Human Interface Device (HID) API

4.3.1 HID Overview

The Human Interface Device class is perhaps the oldest and most established USB device class, in that it was originally created for mice and keyboards. It also supports a wide variety of other “PC peripherals” that consist primarily of various “controls” (buttons, joysticks, volume knobs, etc.).

Like CDC and MSC, HID is supported natively in any common host operating system. This has great advantages, discussed below. But whereas CDC/MSC generate interfaces on the host that are extremely common and popular (virtual COM port or storage volume, respectively), the manifestation of an HID interface on the host is more specialized. For many host programmers, then, it represents a small learning curve. To help with this, TI makes available the *Windows HID API,* which is designed to pair with the MSP430 HID API stack.

A unique aspect of host interaction with HID devices is that often the OS itself is the application interfacing with the device, as in the case of mice and keyboards. In other cases – usually in more general-purpose applications – regular applications interface with the device, just as with CDC.

Some legacy HID implementations request the host to send data via *control endpoint zero.* Usually, the host only uses this endpoint only for management functions. This feature isn’t supported in this API, because the MSP430 USB module has plenty of other endpoints available, and also because its endpoint zero is only eight bytes wide (compared to 64 for the others).

There is a sub-protocol in HID called the boot protocol. A host PC can run this protocol from its BIOS program, prior to the full operating system loading its HID drivers, allowing mice and keyboards to operate more quickly when booting a PC. This protocol is not supported by this API.

The HID API stack supports two kinds of devices. One is for creating “traditional” HID devices. The other is for creating “datapipe” HID devices – a way of adapting HID for general purpose use. It eliminates complexities associated with HID, making it as easy to use as CDC. The next section discusses this in more detail.

4.3.2 HID-Datapipe Approach

The HID protocol is built around a data element called a *report.* Reports can be defined for both the IN and OUT directions (in and out of the host). There are also multiple kind of reports

Report formats are extremely customizable, using a USB descriptor called a *report descriptor.* Report descriptors essentially have their own scripting language, capable of creating an amazing variety of formats. The organization is not flat; it is filled with concepts such as *collections* and *usages,* which can be placed inside each other to several levels.

HID report formats are complex and require a learning curve. They're useful when interfacing with the host OS, since it has no other way of knowing the device's data format. But when the same organization creates both the device and host software, this is not a problem; they are free to assign whatever format they want, and don't need complex HID report formatting to accomplish it. A simple unformatted datastream, like a COM port, becomes sufficient, and the application can implement its own protocol and format.

The HID datapipe function calls implement an unformatted datastream, just like the CDC calls. It defines a very simple report, consisting of a size byte and a large data field. At the level of the MSP430 application, there's essentially no difference in how the CDC and HID-Datapipe calls operate. The call sets are identical, except for their prefix (*USBCDC_* vs. *USBHID_*). Programs written for one can be quickly migrated to the other. The engineer who learns one of these APIs already knows the other.

The datapipe calls are not an industry standard protocol, but rather something created by TI. However, this type of approach is commonly used among USB engineers.

The HID-Datapipe function calls are those described in Sec. 10.4.

4.3.3 HID-Traditional Approach

For engineers that need to create a traditional HID device – that is, one with a customized report descriptor – there is complete freedom and flexibility to do so. Some engineers need this in order to use existing host HID applications, or need to allow the host OS to serve as the HID "application", requiring a particular report format.

Sec. 9 discusses how to implement a traditional HID device using this API.

4.3.4 Host Considerations

The HID API Stack has been tested with Windows; MacOS, and Linux.

4.3.4.1 Advantages of Native Host OS Driver Support

Like CDC and MSC, the HID protocol is supported natively by the major operating systems. This has great advantages:

- Less hassle for the OEM (no need to prepare a kernel-mode driver installation)
- Less hassle for the end user (doesn't have to perform one)
- Problems are less likely to occur; leading to greater stability and lower support costs

4.3.4.2 Microsoft Windows

The HID class is supported by Windows 2000, XP, Vista, and 7.

Unlike CDC (but like MSC), HID devices always load silently onto Windows hosts. They also load silently on Mac/Linux, without the help of an INF file. This is much simpler and troublefree for the end user.

A host application is required. Unlike CDC, which can be used with ordinary "terminal" applications, a HID device generally requires a custom application (unless it's of a type recognized by the host OS). TI provides the *Windows HID API* to assist with this. This API is specifically designed to support the HID-Datapipe model; it uses the same report format, and its calls are designed for moving an unformatted datastream. However, it could be adapted for customized reports (traditional HID) as well.

The demo application provided with the Windows HID API is *HID Demo App.exe.* This executable is provided with the HID application examples accompanying the API source.

The HID API stack has been tested on Windows XP Service Pack 2, Windows XP Service Pack 3, and Windows Vista.

4.3.4.3 MacOS

The HID class is supported on any version of the MacOS. The HID API stack has been tested with version 10.5.6.

4.3.4.4 Linux

The HID class is supported on any recent, common Linux distribution. Enumeration of the HID API stack has been tested with Ubuntu version 9.04, but data transfer has not been fully tested yet.

4.3.5 Transmission Speeds

HID interfaces use USB *interrupt transfers*, compared to CDC's *bulk transfers.* From this they receive guaranteed bandwidth, even on a busy bus; but they're limited in bandwidth to 64KB/sec.

To achieve maximum bandwidth, it's important for the polling interval to be configured as 1ms. This can be done with the Descriptor Tool (see Sec. 5). This is interval at which the host will exchange data between itself and this interface.

If 64KB/sec isn't quite fast enough, one possibility is to create multiple HID interfaces in composite. The Descriptor Tool can easily create a device with several HID interfaces. The MSP430 and host applications could then both be written to interleave their data across them.

Although the bandwidth isn't large, it's immune to bus loading effects. A heavy bus isn't capable of changing it.

4.4 The Mass Storage Class (MSC) API

4.4.1 MSC Overview

An MSP430 running this API will be seen by the USB host as a storage volume. Storage volumes are supported natively by practically any host operating system. Native support has significant advantages, discussed in Sec. 4.4.3 below.

MSC devices are sometimes called "MSD" (Mass Storage Device). They refer to the same USB device class.

This API implements the Mass Storage Class, as specified by the USB Implementers Forum. It specifically implements the *bulk-only transport* (BOT) protocol, rather than the *control/bulk/interrupt* (CBI) protocol. The latter is only intended for legacy applications.

A primary purpose of the MSC protocol is to receive and execute *SCSI commands* from the host. SCSI (Small Computer System Interface, pronounced “scuzzy”) is a set of specifications covering various levels of protocol, including a physical cable interface. One of the SCSI command sets – the *SCSI transparent command set* – has been adopted for use with the MSC protocol. This is the command set supported by this API.

The actual commands supported are shown in Appendex B. Basically speaking, the commands include the ability to determine the device’s identity (like USB descriptors, except on a storage-device level); read/write capability; and error detection and handling.

All handling of SCSI commands is performed automatically by the API, with some support by the application.

The API implements what is known as a *direct-block access device.* This means it uses the subset of SCSI commands called the *SCSI Block Commands (SBC).* Most USB flash-based storage applications use this set, including standard “thumb” flash drives and removable-media flash formats like SD card, and MultiMedia Card (MMC).

This version of the API does not support the *MultiMedia Command set* (MMC). (This is not to be confused with MultiMedia *Cards,* which are a completely separate concept.) The MultiMedia Commands are used to implement CD-ROM and DVD players.

Unlike CDC/HID, the MSC API can only create one MSC interface within a physical device. (It can still contain multiple CDC/HID interfaces in composite with that MSC interface.) If multiple storage volumes are desired, this can be accomplished with multiple *logical units (LUNs).* The API supports any number of LUNs.

4.4.2 File Systems

In a standard implementation of an MSC device, file system software is required so that the MCU application can access its own local storage volume. (This is discussed in more detail in Sec. 8). This API doesn’t integrate a file system; choosing a file system is the responsibility of the software developer using this API. The API is designed to be usable with any file system the developer chooses. Many third party and open source options are available.

However, the application examples accompanying this API include a port of a popular open-source file system, which the developer is free to use. Also, one of the examples shows a form of “file system emulation”, which allows host access to the volume without a file system, at the expense of restricting it to a single file.

When file system software is written, it is designed to support a particular file system format. The storage volume (stored in internal flash or an external medium) must be of a file format comprehensible by both the host OS and the MCU’s file system software. There are several of these standards available, but one that is nearly universal is the FAT file system. FAT has been around since MS-DOS days and is fully supported by Windows, MacOS, Linux, and most other OSes. The MSC API is primarily designed for use with FAT file systems.

4.4.3 Host Considerations

The MSC API Stack has been tested with Windows; MacOS, and Linux.

4.4.3.1 Advantages of Native Host OS Driver Support

Like CDC and HID, the MSC protocol is supported natively by the major operating systems. This has great advantages:

- Less hassle for the OEM (no need to prepare a kernel-mode driver installation)
- Less hassle for the end user (doesn't have to perform one)
- Problems are less likely to occur; leading to greater stability and lower support costs

4.4.3.2 Microsoft Windows

The MSC class is supported by Windows 2000, XP, Vista, and 7.

Unlike CDC (but like HID), MSC devices always load silently onto Windows hosts, without the help of an INF file. They also load silently on Mac/Linux. This is simple and troublefree for the end user.

A host application is required. Any existing application designed to interface with storage volumes can communicate with an MSC interface. If instead a new host application is to be developed, it's relatively simple, because file access is one of the most common functions performed in software. This means it's very well-understood by programmers, and there's a large amount of sample code available in the public domain.

The MSC API stack has been functionally stress-tested with a storage volume benchmarking program, on Windows XP Service Pack 2, Windows XP Service Pack 3, and Windows Vista.

4.4.3.3 MacOS

The MSC class is supported on any modern version of the MacOS. The MSC API stack has been functionally tested with version 10.5.6.

4.4.3.4 Linux

The MSC class is supported on any common Linux distribution. The MSC API stack has been functionally tested with Ubuntu version 9.04.

4.4.4 Transmission Speeds

Full-speed USB is rated 12Mbps. This is a theoretical maximum, and it includes protocol overhead – so it isn't possible for a practical application to achieve this rate for data payload. Further, USB is a system involving many components. Any of these components is capable of reducing the bandwidth.

The following factors can affect bandwidth:

- Host application. USB is very host-driven, which means the host initiates all data transfers, whether sending or receiving. If the host application doesn't initiate transfers often enough, data will be slower.
- Bus loading. MSC uses bulk transfers, which have the potential to reach the highest data rates by using spare bandwidth on the bus. However, the tradeoff to this is that bandwidth scarcity can cause transfers to slow.
- Software on the USB device. If the bus is fast (that is, if the factors mentioned above aren't limiting), the device's application will become the bottleneck in the system. The application requires CPU capacity in order to process data received and to prepare data for sending.

If bandwidth is a priority, DMA should be used. (This can be enabled using the Descriptor Tool.)

5 MSP430 USB Descriptor Tool

The discussion in this section applies to any implementation with the MSP430 USB API stacks (CDC, HID, and MSC).

5.1 What is the Tool?

The Descriptor Tool generates a set of *USB descriptors* and configures the API stack's operation. Simply speaking, it serves three functions:

1. Builds the *USB interfaces* the vendor wishes to support.
2. Configure application-specific MCU settings (clock speed, DMA channel selection, etc.)
3. Customizes the *USB descriptors* with identification data (vendor- and product-specific), which should be done for every new USB device

5.2 What is an "Interface"?

A USB *interface,* in simplified terms, is a stream of data between the USB device and the host. An interface is defined to be of a particular device class, like CDC or HID. It's associated with a specific kind of coding interface on the host – for example, CDC is associated with a virtual COM port.

The USB descriptors declare to the host what interface(s) it supports. Most devices contain only a single interface, but some contain more; these are called *composite* USB devices. The Tool can quickly build any set of interfaces that fit the application.

5.3 What are USB Descriptors?

Every USB device contains *descriptors*, which communicate the device's "identity" to the host. This includes which USB interfaces are supported, as well as other capabilities. They're defined in the files *descriptors.c/h*.

Another set of important information in the descriptors is vendor-specific identification information, including the vendor ID and product ID (VID/PID), and also several strings that may be displayed by the host in connection with this device. The VID/PID combination serves as a identifier for a given product, and therefore it's important that it be unique to this particular product. (When developing a USB device, it's recommended to change the PID any time the USB descriptors change in any way, since most hosts keep archived information about a device, indexed by the VID/PID.)

For more information about descriptors, VIDs, and PIDs, please see the application note *Starting a USB Design with MSP430 MCUs.*

5.4 When to Use the Tool?

The Descriptor Tool should be run once before beginning development, to accomplish #1 and #2 above. #3 can also be done at the same time, if the values are known; or the Descriptor Tool can be re-run at any time later in the development process.

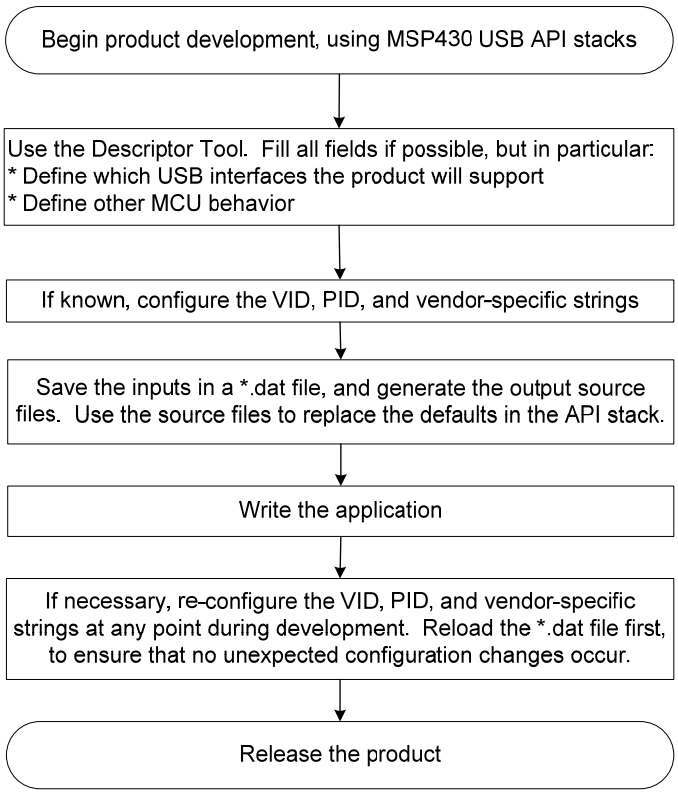


Figure 3. Development Flow Using the Descriptor Tool

5.5 Using the Tool

The Tool can generate descriptors for any combination of USB interfaces – whether single-interface or composite. Unlike creating them manually, it does so reliably and on the first try – saving valuable development time.

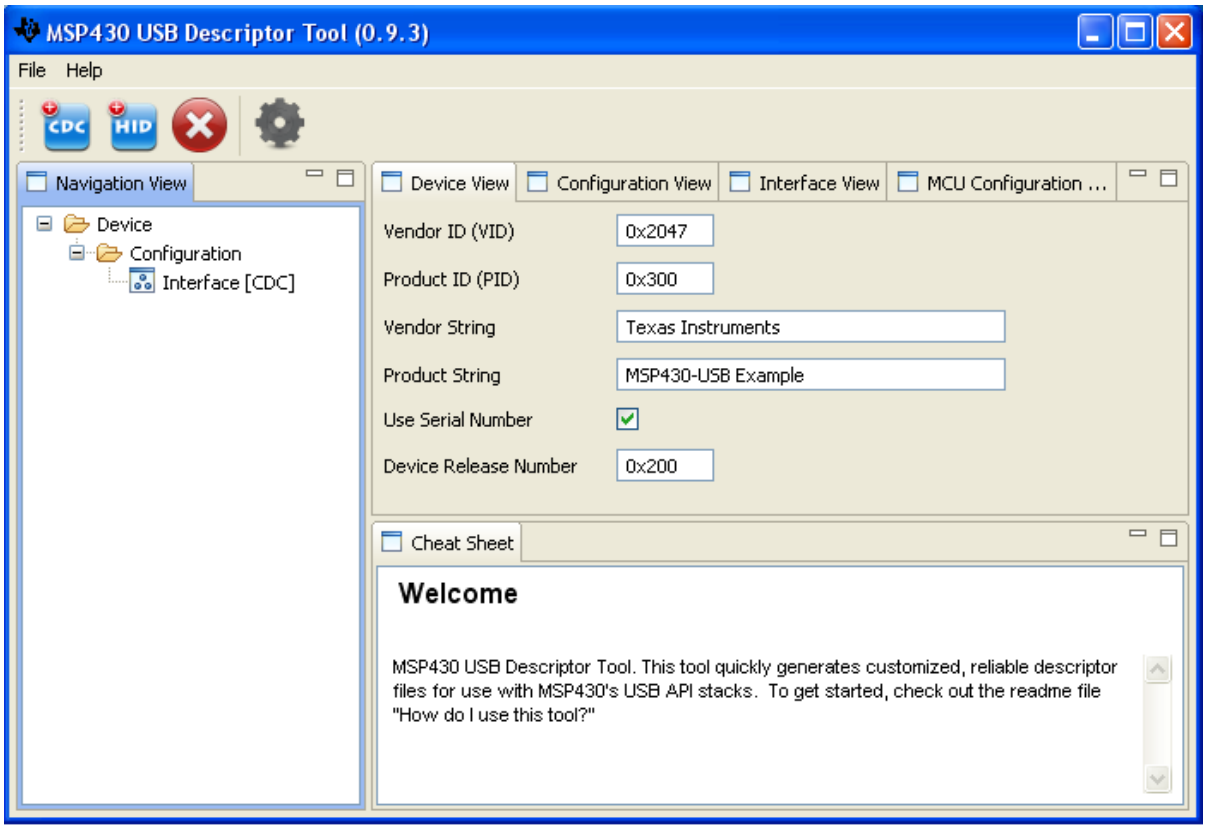


Figure 4. The MSP430 USB Descriptor Tool

In the navigation view, the interface structure can be configured. (The one in the figure shows a single-interface CDC device, sufficient to generate a single COM port on the host.) Clicking the CDC/HID buttons creates additional interfaces to this device, while the "X" button deletes them.

As discussed in Sec. 4.2, a CDC interface results in a COM port being generated on the host, and an MSC interface (if coupled with application code that accesses media) results in a storage volume being generated on the host.

The HID interface implemented by the Tool is for HID-Datapipe, rather than HID-Traditional. HID-Traditional, by definition, requires the engineer to write a custom HID report descriptor. It's recommended to use the Tool to generate a HID-Datapipe interface, then modify the *descriptors.c/h* output files. (See Sec. 9 for more information.)

In the tabbed views, information necessary to build the descriptors can be entered.

The "cheat sheet" is a help pane that provides real-time support for each field in the tabbed views. They explain the tradeoffs and impacts of each decision.

When all the information is input, pressing the "gear" button generates new code files, as described in the next section. After generating the files, it is recommended to also save a *.dat file, so that the same output can easily be generated again at a later time.

5.6 The Tool's Generated Output

The Tool outputs four files:

- *descriptors.c*
- *descriptors.h*
- *usbisr.c*
- If one of the interfaces is CDC, an INF file (**.inf)* is generated, for device installation on Windows hosts

Descriptors.c/h contain the actual descriptor code, as well as structures that help the API know what kind of data interfaces it should make available to the MSP430 application.

usbisr.c is also generated, because the Tool needs to re-arrange the usage of USB endpoints according to what interfaces were created. The USB ISR has a handler for each endpoint, and modifying the ISR ensures that the right endpoint is associated with the right interface.

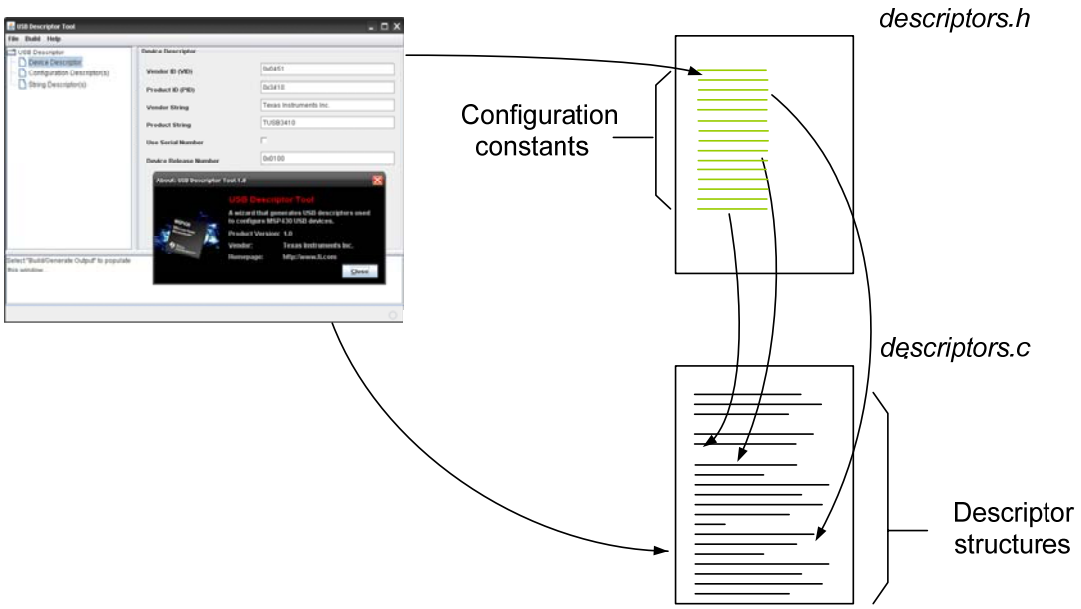


Figure 5. The Descriptor Tool's Output

The INF file requirements can change with the chosen USB interface configuration. For example, the INF file for a device with a single CDC interface is different from one for a CDC+HID device. The Tool generates the correct one for the chosen interfaces, drawing from the same string and VID/PID information that was used to form the USB descriptors.

descriptors.c/h and *usbisr.c* need to be placed into the API stack, replacing the defaults. The API stack provides a directory especially for these files: \USB_config.

It is recommended to use the “Rebuild all” option the first time compiling after replacing the default files with Descriptor Tool output files. This is because the development environment may not realize these files have changed.

5.7 Accessing Interfaces from the Application

It’s important the host and MSP430 application each understand how a device’s interfaces are mapped, for proper communication. The Descriptor Tool helps with this. In the figure below, three interfaces are defined: two CDC and one HID.

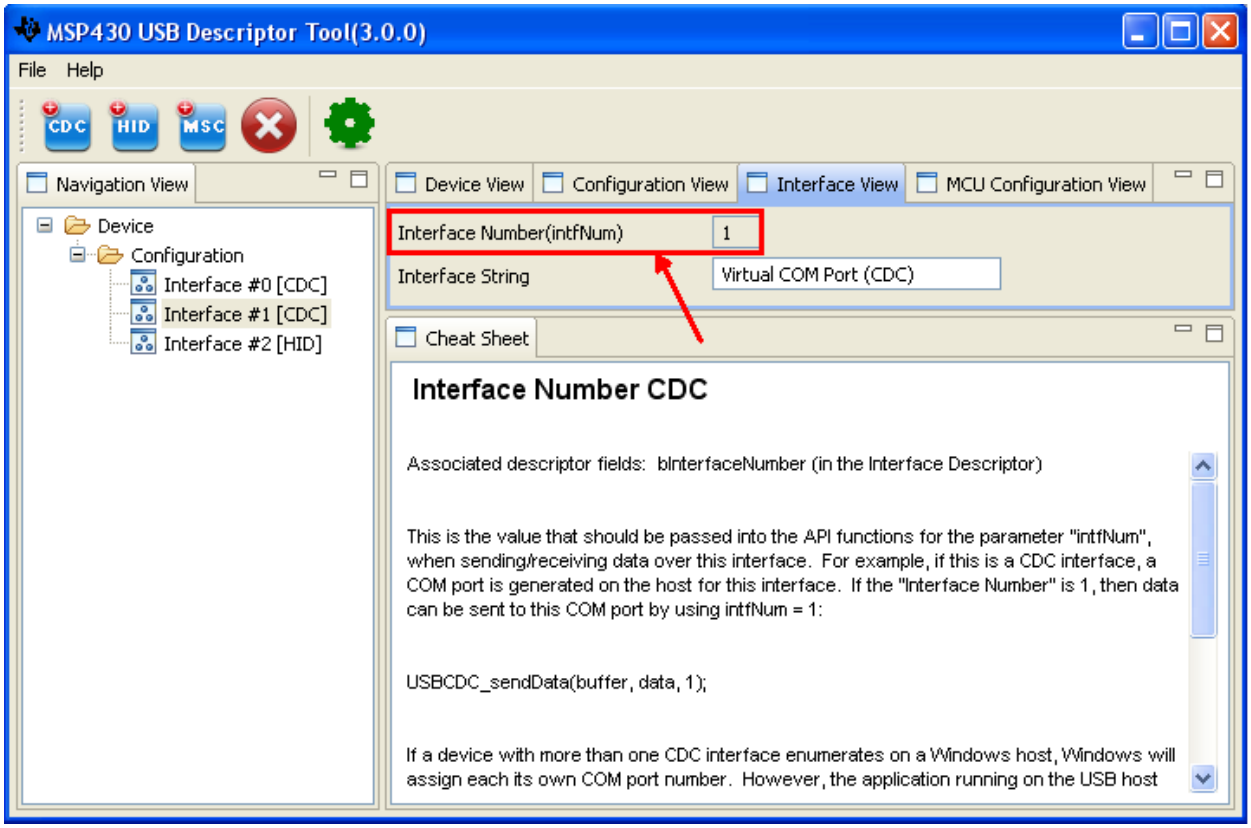


Figure 6. intfNum in the Descriptor Tool

With this done, the stack is then configured to have these interfaces. The application can access them using API function calls containing a parameter called *intfNum.* The value of *intfNum* for each interface is displayed within the Descriptor Tool, as shown in the figure.

If only a single interface is created, *intfNum* is always zero. The interfaces (and therefore *intfNum* values) are always grouped as MSC first, then CDC, and finally HID.

5.8 USB Configurations

The API and Descriptor Tool support only a single USB configuration. (The term *USB configuration* has a specific meaning within the USB specification. Essentially, it refers to a set of USB interfaces that are in effect at any given time).

6 USB States/Events and How They Relate to the API

Every USB device passes through a set of states that describes its interaction with the host. These are shown below. The diagram makes references to function calls within the API, described later in this document.

The discussion in this section applies to any USB device (CDC, HID, and MSC).

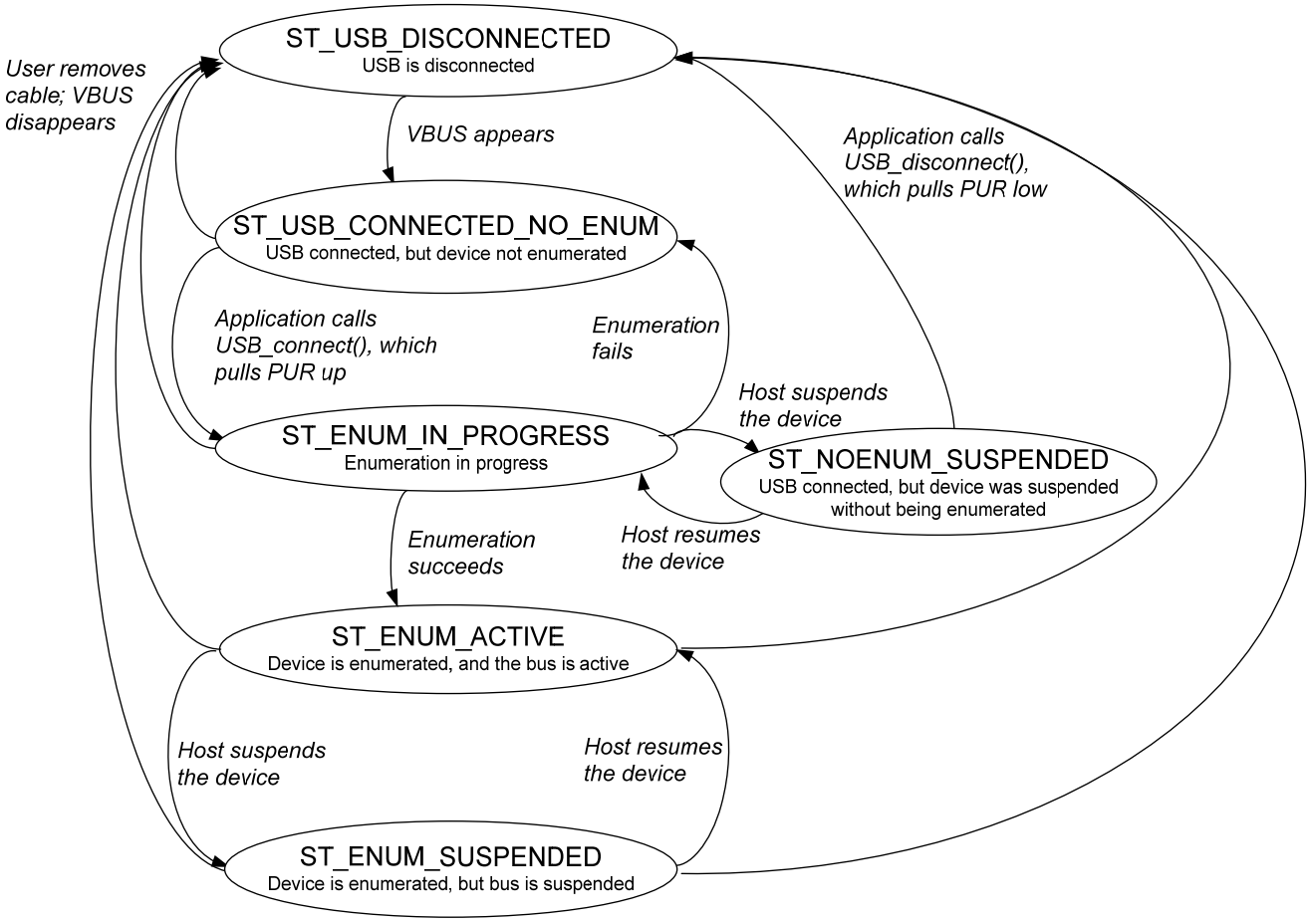


Figure 7. State Diagram Reflecting a Device's Interaction with USB

The state of the connection is returned by the function *USB_connectionState().* The underlying information that defines the state can be returned with the function *USB_connectionInfo().* The relationship between these are shown in the table below.

Table 3. USB State Definitions

	USB_connectionInfo()				
USB_connectionState()		VBUS detected?	PUR high?	Enumerated?	Suspended?
	ST_USB_DISCONNECTED				
	ST_USB_CONNECTED_NO_ENUM	X			
	ST_ENUM_IN_PROGRESS	X	X		
	ST_ENUM_ACTIVE	X	X	X	
	ST_ENUM_SUSPENDED	X	X	X	X
	ST_NOENUM_SUSPENDED	X	X		X

Although each of these factors is discussed in the sections below, here is a brief description:

- *VBUS*: 5V power from the host. If present, the application can assume a host is attached.
- *PUR*: PUllup Resistor. A USB device signals its presence to the host by pulling up the D+ signal through a resistor. The MSP430 implements this pullup with the PUR pin, controlled by software.
- *Enumerated*: When a USB device has been successfully enumerated, it means the host has successfully interpreted the device's descriptors and loaded the appropriate driver. The process requires a series of USB device requests to complete.
- *Suspended*: A host can suspend a USB device at any time, at which point no communication can take place, and current allowed to be drawn from 5V VBUS is restricted

Most applications are best served by using *USB_connectionState()* to direct program flow. A device is likely to spend most of its time in the disconnected, active, and suspended states.

The sections that follow elaborate on topics related to the state diagram.

6.1 Initializing the API

Before any other API call, the application must call *USB_init().* It is recommended to call *USB_setEnabledEvents()* at the same time to enable any events the application will later use.

6.2 Detection of the Host via VBUS

A device can usually know an active host is present by sensing the availability of 5V on the VBUS signal.

Note: Some hardware situations apply 5V to the VBUS pin even though there is no host attached. Sometimes this is done intentionally to take advantage of the integrated LDO, or sometimes the device's upstream USB connector is attached to a powered hub without a host. In these events, *USB_connectionInfo()* will show VBUS on, but *USB_connectionState()* will return ST_NOENUM_SUSPENDED. The application must handle accordingly.

When VBUS transitions on or off, an API event is generated. These events are handled by *handleVbusOnEvent()* and *handleVbusOffEvent(),* respectively. To be used, these must first be enabled with *USB_setEnabledEvents().* They can be thought of as interrupts, and in fact are derived from the USB interrupt service routine handler.

There are various ways to use these mechanisms. The recommended way is to put code in *handleVbusOnEvent()* that connects to the host, since this is the behavior usually expected of a USB device when VBUS appears. Another method would be to poll *USB_connectionState()* from within a main loop, and connect to the host if the returned state is ST_USB_CONN_NO_ENUM.

6.3 Connection to the Host

Connecting to the host usually consists of subsequent calls to *USB_enable(), USB_reset(),* and *USB_connect():*

```
if (USB_enable() == kUSB_succeed)
{
  USB_reset();
  USB_connect();
}
```

6.4 Enumeration

A full-speed USB device makes its presence known to the host by activating a pull-up resistor on the D+ signalling line. MSP430 integrates this with the PUR pin. As mentioned above, the default `handleVbusOnEvent()` handler activates this pullup by calling *USB_connect().*

When the host sees this pull-up, it begins the *enumeration* process, by which it polls the device and loads it onto the system. The API handles enumeration automatically. As it does so, it makes use of the descriptor information in *descriptors.c.*

When enumeration is complete, a call to *USB_connectionState()* reflects this, and the system is ready to transfer data. A `handleEnumCompleteEvent()` is also generated. (The API determines the completion of enumeration by the point at which the host sends a SET_CONFIGURATION request.)

The enumerated state ends when VBUS is removed (state automatically reverts to ST_USB_DISCONNECTED and a `handleVbusOffEvent()` is generated), or when a call is made to *USB_disconnect()* or *USB_disable().*

6.5 Suspend/Resume

At any point after successful enumeration, the host may choose to *suspend* the device. This is characterized by 3ms of inactivity on the data signals (D+/D-). Once the USB device recognizes this event, it has seven more milliseconds to move into a state where it consumes minimal current from VBUS. After this, it cannot communicate with the host until the host resumes it.

Power management of the MSP430's internal USB functionality is handled automatically by the USB module and API. They disable the PLL and shut down much of the USB circuitry, only keeping active what is necessary to detect a resume event (that is, when the host begins communicating on the data signals again). With the PLL disabled, the USB module becomes clocked by the MSP430's VLO oscillator (low-frequency, low-power).

With the PLL no longer using XT2 as a reference, XT2 can be disabled during suspend. If selected in the Descriptor Tool, the API will attempt to do this. However, keep in mind that if any peripherals have selected XT2 as their clock source, then XT2 will remain enabled due to those peripherals issuing it a "clock request". The current draw required for XT2 should be considered in the VBUS budget during suspend.

A host's decision to suspend is largely based on its own activity state and its sensitivity to power drain. Desktop PCs are likely to keep the device active for a long period of time, and only suspend when the PC itself enters a powerdown state. Laptops are similar, except of course they tend to enter a powerdown state more often, due to being battery-powered. Mobile hosts may cut power even more frequently.

Although the internal circuitry's power is handled automatically, the application may need to take steps to reduce the overall system's power draw during suspend. See Sec. 12.1 for information.

6.6 Remote Wakeup

A remote wakeup event is a mechanism by which a suspended USB device can prompt the host to resume it, perhaps waking the host in the process. A common example of remote wakeup is when a USB mouse is attached to a PC, and the PC goes into standby mode. Some configurations allow the mouse to wake the PC when moved, by issuing a remote wakeup event. After waking, the host resumes the mouse.

A device first must declare itself as capable of remote wakeup within its configuration descriptor. The Descriptor Tool can configure the API to do this. The host may choose to grant the ability for remote wakeup, or it may choose not to. Whether it does so is OS-dependent and may also be dependent on user configuration.

The application can issue a remote wakeup using the *USB_forceRemoteWakeup()* function. If it returns *kUSB_succeed*, it means the host indeed had enabled the remote wakeup function and may choose to respond to the request by resuming the device. A resume will be evident to the application through means of a return to the ST_ENUM_ACTIVE state, and the *handleResumeEvent()* handler will execute, if enabled. If *USB_forceRemoteWakeup()* returns with *kUSB_generalError*, it means the host did not enable remote wakeup for this device.

6.7 Failed Enumeration

If the device pulls PUR high and it does not lead to successful enumeration, *USB_connectionState()* will stay in the ST_ENUM_IN_PROGRESS. Depending on the reason for the failed enumeration, the host may then suspend the device, which puts it into ST_NOENUM_SUSPENDED.

6.7.1 Attachment to Powered Busses that Lack an Active Host

Two possible ways of entering ST_NOENUM_SUSPENDED include:

- the device attempted to enumerate while attached to a powered hub that doesn't have a host upstream from it; or
- the device attempted to enumerate on a host that is indeed powered, but in a 'standby' mode (during which all USB devices are suspended)

The only way a USB device can detect the presence of a host using static signals is the presence of VBUS. However, this doesn't guarantee that the host is truly present and active. The only way to be certain is to attempt enumeration by pulling PUR high (calling *USB_connect()*). If the host doesn't respond by attempting to enumerate within 3ms, the MSP430 will recognize it as a suspend event. The combination of having attempted to enumerate, but being suspended before completion defines the ST_NOENUM_SUSPENDED state.

The software designer can choose to stay in this state until the bus situation changes; or if the situation is prolonged, disconnect from USB (de-assert PUR). The bus situation could change if, for example, the host awoke from standby and resumed all its USB devices.

If the application chooses to disconnect from USB, the device will not be able to recognize a host attempting to resume it. Therefore, the end user would need to detach and re-attach the device before it could be enumerated.

If instead the application chooses to stay in ST_NOENUM_SUSPENDED, then this state must be handled according to how the device should function while waiting for the host to begin enumeration. This state keeps the USB module active, allowing it to detect a resume event. The API would then automatically handle the event, execute the enumeration process, and bring the device into ST_ENUM_ACTIVE.

A possible way to implement this is as follows:

```
case (ST_USB_DISCONNECTED || ST_NOENUM_SUSPENDED):
   __bis_SR_register(LPM3_bits + GIE);
   break;
```

This code treats a ST_NOENUM_SUSPENDED device the same as if disconnected. The only difference between the states is that in the latter, the USB module is enabled (but suspended). (This also means the device is consuming somewhat more current.) If the host later resumes the device, the device's USB module is kept on to detect it, and the resume event will cause the state to become ST_ENUM_IN_PROGRESS.

6.7.2 Active Host Attempts Enumeration, but It Fails

If enumeration begins but fails for some reason, the host may suspend the device. This would put it into ST_NOENUM_SUSPENDED.

6.8 Removal from the Bus

When the end user detaches the device from the host, this is recognized by the device as a VBUS-off event (bus power is no longer available on the VBUS pin). The API responds by disconnected from USB and disabling the USB module. It also calls *handleVbusOffEvent()*. After this, a call to *USB_connectionState()*will return ST_USB_DISCONNECTED.

As with suspend events, it is very important that the software designer consider that the end user may remove the bus at any moment during execution. This is sometimes referred to as a "surprise removal". Software must anticipate that this and be able to recover gracefully.

6.9 USB Hardware Conditions in Each State

Each state is associated with certain hardware conditions, as shown below.

Table 4. USB State Hardware Conditions

	USB Module Enabled	USB Transceiver[1]	Integrated LDOs[2]	XT2 Oscillator	PLL Enabled
ST_USB_DISCONNECTED					
ST_USB_CONNECTED_NO_ENUM	X	Idle	X	X	X
ST_ENUM_IN_PROGRESS	X	Active	X	X	X
ST_ENUM_ACTIVE	X	Idle/Active	X	X	X
ST_ENUM_SUSPENDED	X	Idle	X	(see note 3)	
ST_NOENUM_SUSPENDED	X	Idle	X	(see note 3)	

Note 1: Idle means the transceiver is powered, but not transceiving any data, which means it is consuming minimal current. Active means data is being transceived, raising the current draw. Idle/Active means the transceiver might be in either condition during this state.
Note 2: The API enables both the 3.3V and 1.8V USB LDOs.
Note 3: XT2 can be kept enabled during suspend, using the Descriptor Tool.

The USB module must be enabled in order to detect suspend/resume events from the host, as well as to transceive data. The other four columns relate primarily to power consumption. In particular, the PLL is a major source of power draw, and must be shut down during USB suspend. Similarly, an active USB transceiver consumes much more current than an idle one. See the device datasheet for specific parameters.

7 CDC and HID-Datapipe: Data Transmission/Reception

The discussion in this section applies to any CDC interface, as well as any HID interfaces using the datapipe function calls. It does not apply to traditional HID interfaces.

7.1 Introduction

The API provides a simple scheme of exchanging data with the USB host using either a CDC or HID interface. This scheme is called the *datapipe*. This is a generic data interface that can be applied to any application. It can send/receive data of any size with a single function call. The API handles all associated USB protocol to accomplish it.

Any CDC interface uses the datapipe. Any HID interface in which the report descriptor has been left as the default can also use the datapipe, by using the HID-Datapipe function calls described in Sec. 9.4. (If the report descriptor has been customized, then the HID-Traditional function calls must be used, described in Sec. 10.5.)

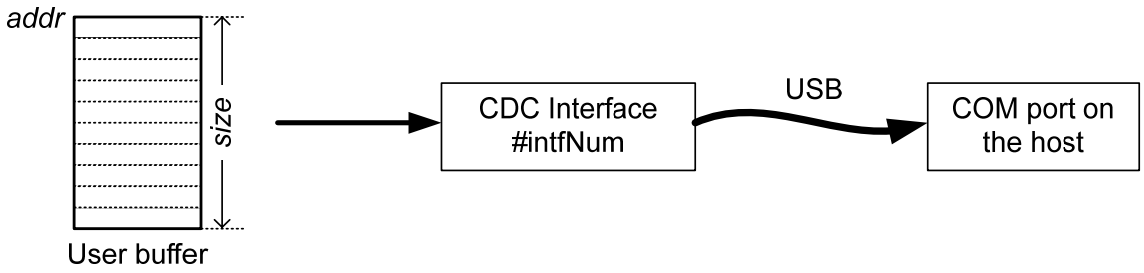
CDC send/receive function calls have the prefix *USBCDC_,* while HID calls have the prefix *USBHID_*. Since the text in this discussion applies equally to CDC or HID-Datapipe, the function calls are sometimes shown to have the prefix *USBxxx_*. The same applies to the prefix for return values (*kUSBxxx_*).

7.2 Send/Receive "Operations"

Send/receive operations are the basis of sending/receiving on a CDC interface or an HID interface using datapipe function calls. This section describes how they work. For practical information and example send/receive construct functions, please see Sec. 12.

For all sending and receiving, the application must first prepare a data source, called the *user buffer*. The application passes the buffer, the number of bytes to be transceived, and the selected interface to *USBxxx_sendData()* or *USBxxx_receiveData().* The user buffer can be located anywhere within the MSP430 memory map.

```
USBCDC_sendData(addr, size, intfNum);
```




```
USBHID_sendData(addr, size, intfNum);
```

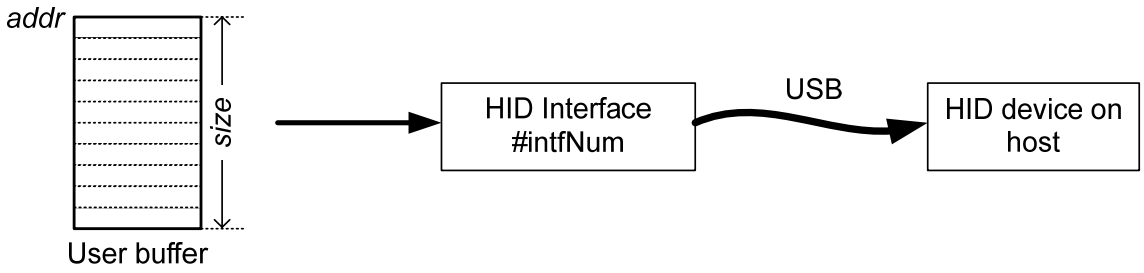


Figure 8. Send Operation

A single call to *USBxxx_sendData()* initiates a *send operation.* The API begins to copy the buffer to the USB endpoint buffer in packetized chunks. After each packet is formed in the endpoint buffer, it's made available to the host. At the host's discretion and timing, it reads the packet from the endpoint buffers. When all the data has been read, the operation is complete.

Similarly, a single call to *USBxxx_receiveData()* initiates a *receive operation.*. The user buffer acts as a data sink to any data subsequently received from the host. As data is received into the USB endpoint buffers for this interface, the API copies it into the user buffer. When the buffer is full, the operation is complete.

7.2.1 User Buffer

The term "buffer" implies a RAM source, but it can also be flash or peripherals – any contiguous block in the MSP430's memory map.

The buffer can be of any size, whether one byte or several kilobytes. All packetization is automatic.

The user buffer is separate from the *endpoint buffers*, which can be thought of as registers within the USB module that store USB data as it waits to be sent to the host, or data that has been received from the host and is waiting to be read by MSP430 software. Unlike the user buffer, the endpoint buffers are limited in size to 64 bytes. The API handles all interactions with the endpoint buffers automatically, so they're not part of the API application programming model.

During a send operation, the data is only copied out of the user buffer; the contents of the buffer remain unchanged. However, care should be taken that the application not write to the buffer until a send operation is known to be complete, as doing so could disrupt the operation.

7.2.2 Background Execution

Send/receive operations are executed “in the background”. For example, when a call to *USBxxx_sendData()* returns, this has no bearing on whether the operation has completed or not -- only that is has started, or will start very soon. The operation will occur in the background as the bus is available, until it’s completed. (In other contexts, this concept is called *asynchronous* operation, because the USB MCU’s execution is “decoupled” from the host, rather than being dependent on it for a quick response.)

This approach has several advantages:

1. Execution is not held in one place while long transfers complete
2. Execution is not held at the mercy of the host/bus’ availability
3. Efficiency is increased, because the API isn’t waiting idle while waiting for the host to fetch the next packet. Other useful activities can be performed during this time.

Internal to the API, a send/receive operation is an interrupt-driven, host-driven process. When the host is ready, it sends/receives data, and this results in interrupts in the MSP430. After the operation is started, the MSP430 resumes execution of the application, and simply responds to these interrupts as they occur. It effectively has “standing orders” to send/receive the user buffer. Once these standing orders no longer apply – that is, once the buffer has been processed – *USBxxx_handleSendCompleted()* and *USBxxx_handleReceiveCompleted()* tell the application that the interface is available for more instructions.

This is directly analogous to sending a block of memory over a SPI interface, using DMA to move the block to the SPI transmit register as the register becomes available. In this example, the DMA module is in control of the operation; in the USB context, the API is in control.

The application can know that an operation has completed by one of two means:

1. A *USBxxx_handleSendCompleted()* or *USBxxx_handleReceiveCompleted()* event occurs
2. A call to *USBxxx_intfStatus()* shows whether or not an operation is still in progress

From the application’s point of view, send/receive operations are automatic; but the software designer should be mindful of the background processing. For example, software should check for the presence of an open operation before starting another one, or before accessing the user buffer of an open operation. That can be done with a call to *USBxxx_intfStatus().*

7.2.3 How Many Operations Can be Open Simultaneously?

An interface may have one send operation and one receive operation in progress at any time – but not more than one of either. If more than one interface exist, each can have their own simultaneous operations.

7.2.4 Behavior During Suspend/Resume

If the device gets suspended by the USB host or if the bus is removed, any active send or receive operations remain open.

7.2.5 Lifecycle of a Send Operation

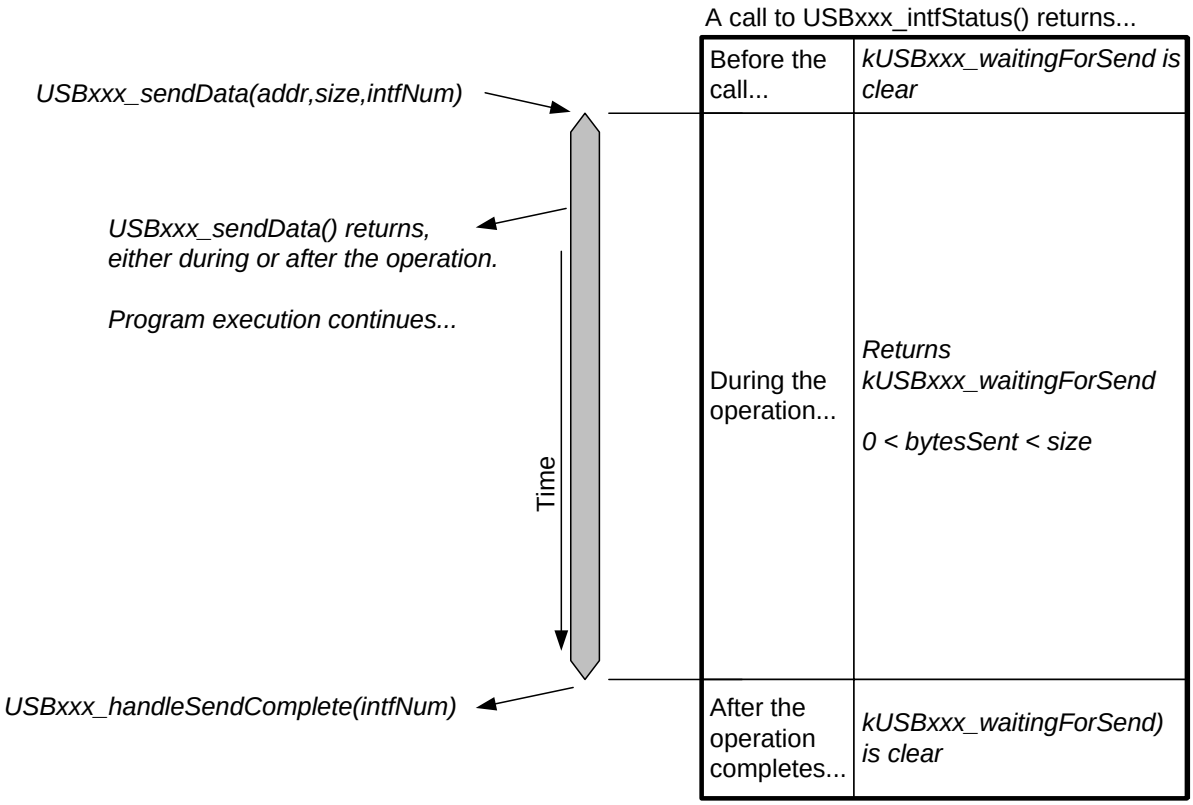


Figure 9. Lifecycle of a Successful Send Operation

As discussed earlier, a send operation is begun with a call to *USBxxx_sendData()*. The application can make its first call to this function any point it wishes while the state is ST_ENUM_ACTIVE. A successful call to this function returns *kUSBxxx_sendStarted.*

If a call to *USBxxx_sendData()* is made while a previous send operation is underway, it will immediately return with a value of *kUSBxxx_intfBusyError.* This is because only one send operation (and one receive operation) may be open for a given interface at a time. The previous operation continues, unaffected.

When a send operation is complete, the API makes a call to *USBxxx_handleSendCompleted().* User code may be placed here, perhaps flagging the application to begin another send operation, or to alert the user that all data has been transmitted.

After *USBxxx _sendData()* returns *kUSBxxx_sendStarted,* software should be aware the operation might still be open. Therefore, any subsequent calls to *USBxxx_sendData()* should check to ensure that no previous operation is underway.

Send operations usually complete fairly quickly, but if an operation is open, it can be aborted with *USBxxx _abortSend().* After aborting the operation, this function returns how many bytes were successfully sent.

7.2.6 Lifecycle of a Receive Operation

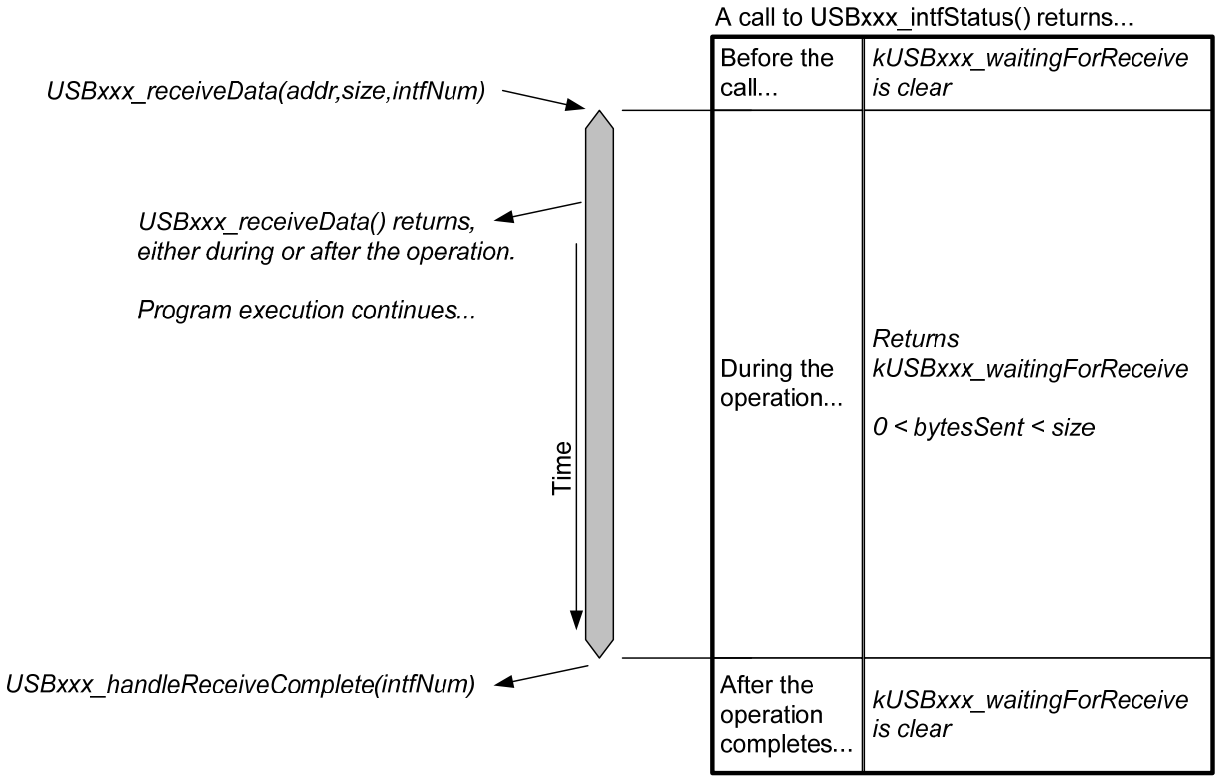


Figure 10. Successful Receive Operation

Similarly, a receive operation is begun with a call to *USBxxx_receiveData()*. The application can make its first call to this function any point it wishes after enumeration is complete. A successful call returns *kUSBxxx_receiveStarted.*

If a call to *USBxxx_receiveData()* is made while a previous receive operation is underway, it will immediately return with a value of *kUSBxxx_intfBusyError.* This is because only one receive operation (and one send operation) may be open for a given interface at a time. The previous operation continues, unaffected.

When a receive operation is complete, the API makes a call to *USBxxx_handleReceiveComplete().* User code may be placed there; for example, it may set a flag that signals main() to begin another receive operation.

If data is received into the USB endpoint buffer without an open receive operation, the API has nowhere to put it. After this, any subsequent attempts by the host to send more data will be NAK'ed by the device, and thus the pipe is effectively "clogged". If this situation occurs, the API makes a call to *USBxxx_handleDataReceived().* This gives the application an opportunity to "unclog" the pipe by either opening an operation or calling *USBxxx_rejectData().* The former gives the incoming data a place to go. The latter flushes the USB endpoint buffer; the "pipe" becomes unclogged again; but the data that was in it is lost.

The function *USBxxx_bytesInUSBBuffer()* can be used to determine how many bytes are waiting in the USB endpoint buffer. This can be useful when the event *USBxxx_handleDataReceived()* occurs; the application can respond by calling *USBxxx_bytesInUSBBuffer(),* and then calling *USBxxx_receiveData()* for the exact number of bytes that are waiting.

7.2.7 How Long Does an Operation Stay Active?

When a send operation is begun, the data is transmitted as quickly as the host, device, and bus conditions will allow. Usually, this is very fast. Obviously, the larger the data size, the longer the transmission takes.

However, any factor that affects bandwidth also has an effect on the duration of a send operation. The bus conditions can potentially have significant ability to delay transactions, and so it's important for software to account for this – including the unknown timelength of open operations.

Send operations are sent as quickly as conditions will allow. Receive operations are subject to an additional factor: depending on the application, it's sometimes unknown when the host will send data. So while send operations are almost assured to happen somewhat continuously, receive operations might be fulfilled in piecewise fashion. If communication conforms to a defined protocol, the application may know when the data is arriving. If this isn't the case, software may need to be written in a more open-ended fashion.

USBxxx_sendData(), USBxxx_receiveData(), and *USBxxx_intfStatus()* have all the return codes necessary to manage this. Also, Sec. 12 provides clear example coding constructs that ensure proper operation.

7.3 Host-Side Considerations When Interfacing to the Datapipe

When using CDC, there are no considerations specifically related to send/receive operations. From the host's perspective, data is simply sent/received through the COM port, as with any other virtual COM port application.

When using HID, the application needs to take care to structure reports according to the API's default report descriptor. The report format is shown below.

Table 5. Reports Described by the Default Report Descriptor

Field	Size	Description
IN report (into the host)		
Report ID	1 byte	The report ID of the chosen report (automatically assigned to 0x3F by the HID-Datapipe calls)
Size	1 byte	The number of valid bytes in the *data* field

Data	62 bytes	Data payload
OUT report (out of the host)		
Report ID	1 byte	The report ID of the chosen report (must be assigned to 0x3F by the host)
Size	1 byte	The number of valid bytes in the *data* field
Data	62 bytes	Data payload

This format effectively converts the report mechanism into a simple data carrier. The MSP430 application sees only an unformatted stream of data. The *Windows HID API* provided for MSP430 does the same thing for Windows applications. A host application for any operating system can accomplish the same thing by formatting data in the manner shown above.

8 MSC: Software Architecture

The MSC API creates an mass storage class USB interface in an MSP430 device – individually, or in composite with CDC and/or HID. This section explains how the developer should build an application around the API.

8.1 MSC Architecture: High-Level Overview

Unlike CDC and HID-Datapipe, an application using an MSC interface doesn't "send" and "receive" data over an unformatted datastream. Instead, it creates a large ("mass") pool of data – a storage volume – to which the host has access. Unless the developer is simply using the MSC interface as a "flash drive" – that is, a storage volume that only the USB host can access – the USB device's application software will also need access to the volume, and this requires file system software.

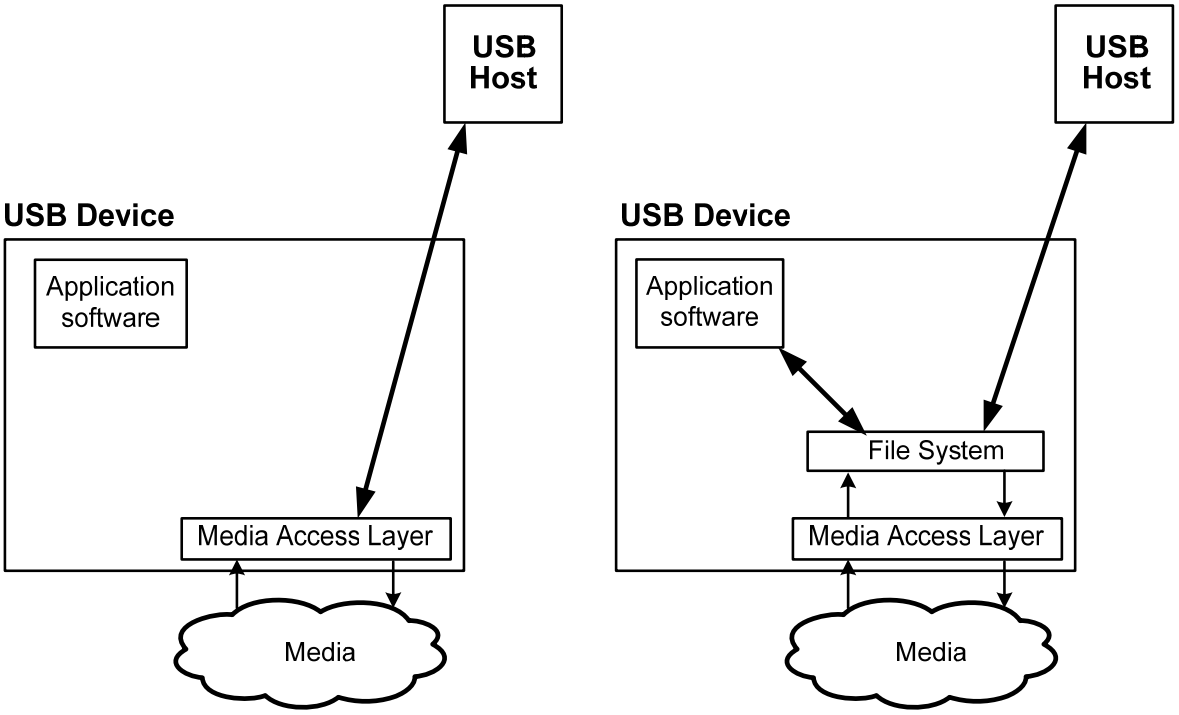


Figure 11. Simplified High-Level MSC Architecture

There are ways around this requirement, although they usually involve tradeoffs. One such way is demonstrated in example application #M1. This example shows a form of file system "emulation", at the expense of only allowing a limited number of files in the volume.

The volume exists on a medium. In an MSP430-based mass storage device, the medium might be internal memory or an external device accessible via the MSP430's interfaces (for example, SPI, I2C, or an emulated parallel memory interface).

8.2 Storage "Address System": LUNs & LBAs

Storage is divided into *logical blocks;* and each is addressed with a *logical block address (LBA).* In the FAT file system, one block consists of 512 bytes. Blocks are sometimes called *sectors.*

The USB host uses LBAs when making requests to the storage device for READ/WRITE operations. Usually the MCU application will pass the LBA to the file system software, when then accesses the volume on the medium. LBAs begin at zero.

If no file system is being used and the medium is a space within MSP430 internal flash, the application might convert the LBA to a byte address simply by multiplying the LBA by the size (in bytes) per logical block, or sector; then add this to any appropriate offset within flash.

All of this occurs within the context of a *logical unit.* Logical units are referenced with a *logical unit number,* or *LUN.* There might be multiple LUNs within an MSC interface. The host operating system (i.e., Windows) presents each LUN to the user as a separate volume. If an equipment maker wishes to have multiple storage media on a single physical mass storage device – for example, a removable media card as well as a separate volume located in internal MSP430 flash – they may choose to implement these as two separate LUNs. In certain respects, this is analogous to having multiple *interfaces* within a composite USB device.

This is illustrated in the figure below.

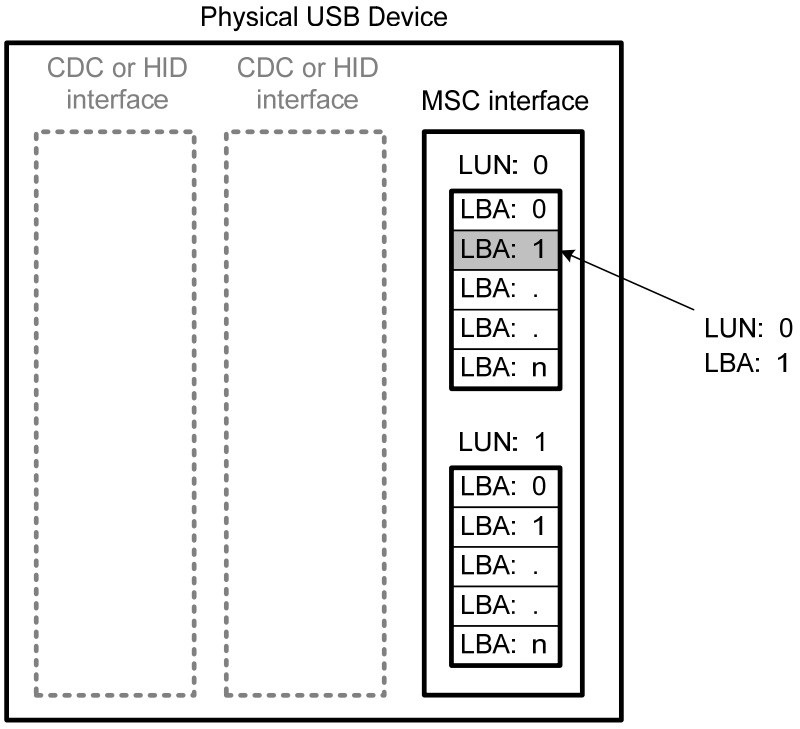


Figure 12. MSC Interface LUN/LBA Addressing

The API supports a multiple LUNs, containing any 32-bit number and size of logical blocks. The number of LUNs can be selected within the Descriptor Tool. (After being selected, each LUN must be implemented within the application.)

When the host sends a SCSI command over the MSC protocol, it designates the LUN for which the command is intended. If the command reads/writes data, it also includes the LBA being accessed and the number of sequential blocks requested. Since the API must rely on the application to make the file system access calls when a SCSI read or write command is received, the API makes the SCSI command's LUN, initial LBA, and number of requested blocks available to the application.

Only one SCSI command at a time can be handled by an MSC interface. This means that if a command is received for LUN 0, it must be handled in full before a command can be received for LUN 1. For this reason, only a single interchange buffer is needed for the interface (or two, if double-buffering is used). (The current API version only supports single-buffering.)

8.3 Components of an MSP430-Based MSC Application

The table below shows the actions the application must take.

Table 6. Actions the Application Must Take

When	Action	How
Initialization	Define the LUN structure and characteristics	*USBMSC_config*
	Allocate the data buffer exchange space, and then register it with the API	*USBMSC_registerBufInfo()*
	Returns the pointer to the API's *USBMSC_RWBuf_Info* structure, which describes any requested buffer operations.	*USBMSC_fetchInfoStruct()*
	Inform the API about each LUN's initial medium	*USBMSC_updateMediaInfo()*
Periodically	Check for any received SCSI commands and initiate their handling	*USBMSC_poll()*
Event-driven	Process any buffer events the API generates during its SCSI READ/WRITE handling	Access the file system, in response to *USBMSC_handleBufferEvent(),* then call *USBMSC_bufferProcessed()*
	If the medium is designated as 'removeable', and the medium changes state, inform the API	*USBMSC_updateMediaInfo()*

Each is defined in more detail below.

8.3.1 Defining the Interface's LUN Organization and LUN Characteristics

An application must define the characteristics of its LUNs. The API then uses this definition when responding to SCSI commands. To accomplish this, the API internally defines a structure type called *config_struct.* The application must define an instance of this, named *USBMSC_config,* that reflects the characteristics of its LUN.

An example for a single-LUN is shown below.

```
// Defined within the application
struct config_struct USBMSC_config =
{
  .number    = 0x00,
  .PDT       = 0x00,
  .removable = 0x80,
  .t10VID    = « TI MSC »,
  .t10PID    = « LUN0 »,
  .t10rev    = « 0x00 »
};
```

An example for two LUNs in shown below.

```
// Defined within the application
struct config_struct USBMSC_config =
{
  {
    .number    = 0x00,
    .PDT       = 0x00,
    .removable = 0x80,
    .t10VID    = « TI MSC »,
    .t10PID    = « LUN0 »,
    .t10rev    = « 0x00 »
  },
  {
    .number    = 0x00,
    .PDT       = 0x00,
    .removable = 0x80,
    .t10VID    = « TI MSC « ,
    .t10PID    = « LUN1 »,
    .t10rev    = « 0x00 »
  },
};
```

The structure's identifiers must be left unchanged, so that the API can interpret them; but the values must be customized. The information to be populated in each field is shown in the table below.

Table 7. LUN Information in the *USBMSC_config* Structure

Field	Format	Description
.LUN.number	BYTE	The logical unit number. (Must be 0x00, because this version of the API only supports one LUN.)
.LUN.PDT	BYTE	Peripheral Device Type. This is a code that identifies to the host which set of SCSI commands to use with this device. (Currently must be 0x00, which is for SBC devices. Most flash-based USB devices use this class.)
.LUN.removable	BYTE	Indicates whether the device's media can be removed – for example, flash media cards. 0x80 indicates the medium is removable; 0x00 indicates it is not.
.LUN.t10VID	BYTE[8]	A vendor ID assigned by the T10 organization.
.LUN.t10PID	BYTE[16]	A product ID assigned by the owner of the T10 VID.
.LUN.t10rev	BYTE[4]	A revision code assigned to a device with this T10 VID/PID.

T10 is the organization that oversees the SCSI specification. (http://www.t10.org) T10 VIDs are freely available. There is no certification process that checks to ensure a unique VID is used, nor does the choice of VID/PID have any significant effect on most USB hosts.

T10 VIDs/PIDs should not be confused with USB VIDs/PIDs; they are not related.

8.3.2 Registering the Location of the Buffer: USBMSC_registerBufInfo()

Exchanging data between the host and the file system requires a memory buffer large enough to hold an entire block. For the FAT file system, this is a multiple of 512 bytes. Since this represents a significant amount of available RAM, the API gives as much control of its allocation to the application as possible. The application must allocate the buffer, and then "register" it with the API. It does this with *USBMSC_registerBufInfo().*

This function passes in three parameters:

- Address of the X-buffer
- Address of the Y-buffer
- The size of the X- and Y-buffers

  Note: The designation of X and Y refers to a double-buffering scheme that is not yet supported in this API. As a result, the API ignores the Y-buffer address.

The application can dynamically change the buffer location, and it can also disable the buffer completely. The latter is advantageous for re-allocating the memory when USB isn't attached. It can be accomplished by calling *USBMSC_registerBufInfo()* with an X-buffer address of *null*. The API always uses the address/size from the most recent call to the function.

If the host attempts to access the MSC interface, but the most recent call to *USBMSC_registerBufInfo()* de-activated the buffer, then the API has no way to exchange data with the application. It begins failing READ/WRITE commands received from the host, telling it that the unit isn't ready. During this time, calls to *USBMSC_poll()* return *kUSB_generalError.*

Therefore, if the buffer is to be dynamically managed, it is strongly recommended to re-instate it in response to *USB_handleVbusOnEvent()* (which occurs when USB is attached). Also, if the buffer is de-activated during USB suspend, it should be re-instated in response to *USB_handleResumeEvent().*

8.3.3 Registering the Buffer Info Structures: USBMSC_fetchInfoStruct()

The API allocates an instance of the structure *USBMSC_RWBuf_Info* to describe any buffer operations it wants the application to process.

The application needs the pointer to this structure, so that it can access the buffer operation description. It must call this function near the beginning of operation, after *USBMSC_registerBufInfo()* but before USB enumeration.

8.3.4 Informing the API About the Media: USBMSC_updateMediaInfo()

For each LUN defined with *USBMSC_config*, the application must describe the storage medium to the API. It must do this in two situations:

1. Initially, before the USB device enumerates (that is, before calling *USB_connect()*).
2. If the media is removeable, it must also inform the API as soon as possible in response to any subsequent changes in the medium.

The function that accomplishes this is *USBMSC_updateMediaInfo()*. With the information provided by this function, the API can respond appropriately to any related SCSI commands from the host.

The application must declare an instance of the API-defined structure *USBMSC_mediaInfoStr* and pass it into *USBMSC_updateMediaInfo().*

Table 8. USBMSC_mediaInfoStr

Type	Name	Description
BYTE	mediaPresent	Indicates that the medium is present (non-zero) or not present (zero).
BYTE	mediaChanged	Indicates that the medium present is a different one than during the last call to *USBMSC_updateMediaInfo()*
BYTE	writeProtected	Indicates that the medium is write-protected (non-zero) or not write-protected (zero).
DWORD	lastBlockLba	LBA of the last block in the media (effectively, the medium's size)
DWORD	bytesPerBlock	Number of bytes per block in this medium (for FAT, this is typically 512)

The last three fields are only valid if *mediaPresent* is non-zero.

If the media is removable (which should be reflected in *USBMSC_config)*, then the application needs functionality to detect it. The means of detection vary, depending on the media type. As an example, SD-card interfaces can detect a pullup in the card, using an I/O pin interrupt.

If the application detects that media has been inserted, it should:

- Create an instance of *USBMSC_mediaInfoStr*
- Set *mediaPresent* and *mediaChanged*
- Determine whether the media is write-protected, and set *writeProtected* accordingly
- Detemine the media's size, and set *lastBlockLba* accordingly.
- Call *USBMSC_updateMediaInfo().*

If instead the application detects that media has been removed, it should:

- Create an instance of *USBMSC_mediaInfoStr*
- Clear *mediaPresent*
- Set *mediaChanged*
- Call *USBMSC_updateMediaInfo().*

To determine the media's size and write-protected status, file system calls are usually available. If necessary, it could parse the volume's master boot record manually.

8.3.5 Periodically Initiating SCSI Command Handling, using USBMSC_poll()

The application must initiate the handling of any SCSI commands that have been received, using *USBMSC_poll().* Any SCSI commands received by the API will not be handled until *USBMSC_poll()* is called. The API does not interrupt the application to tell it commands have been received; rather the application needs to call *USBMSC_poll()* periodically.

8.3.5.1 Main Loop, No LPM Mode

Many USB devices derive their power from the host (VBUS) while enumerated and active, and so they don't have tight power requirements. Developers of these devices might choose not to enter a low-power mode.

If the application is based on a main loop, and doesn't enter LPM0, the calling of *USBMSC_poll()* can simply be placed within the main loop. Note that if the function is called when there is no SCSI command to handle, it will quickly return with no action taken.

8.3.5.2 Main Loop, Entering an LPM Mode

The API is designed for LPM0 to be entered within a main loop structure. If a SCSI command occurs while the CPU is in LPM0, it automatically wakes the CPU, allowing the application's main loop to call *USBMSC_poll().*

Most SCSI commands are handled automatically when *USBMSC_poll()* is called, without any help needed from the application. By the time *USBMSC_poll()* returns, the command has been handled. However, SCSI READ/WRITE commands require the application to "process" buffers. When this occurs, the API automatically wakes the CPU out of LPM0, and execution resumes from its point of entry.

Given this, the following coding structure is recommended in applications entering LPM0:

```
__disable_interrupt();
if(USBMSC_poll() == kUSBMSC_okToSleep)
{
  __bis_SR_register(LPM0_bits + GIE);
}
__enable_interrupt();
```

This structure ensures robust handling. The condition it guards against is one in which the CPU enters LPM0 even though the API is waiting for it to process a buffer. It does this by first disabling interrupts, to prevent the API from servicing any subsequently-received SCSI commands. *USBMSC_poll()* then checks to see if any SCSI commands have been received up to that time. If not, it returns *kUSBMSC_okToSleep.* Then, atomatically with the LPM0 entry, it re-enables interrupts. This way, if a SCSI READ/WRITE command was received during *USBMSC_poll(),* the application won't miss the fact that it's supposed to remain awake; instead, the API will begin processing it immediately after the LPM0 entry, and the application will immediately wake again.

Alternatively, if a READ/WRITE command was already received and the API is waiting for the application to process a buffer, *USBMSC_poll()* will return *kUSBMSC_processBuffer,* keeping the loop awake.

Note that if LPM0 is not being entered – rather the CPU will be kept continuously active during the USB connection – then it is not necessary to disable interrupts or check the return value from *USBMSC_poll().*

8.3.5.3 Frequency of Calling USBMSC_poll()

It is important that *USBMSC_poll()* be called with sufficient frequency. There are two main concerns the developer should consider:

- Slow "average" polling frequency, leading to slow bandwidth performance
- An occasional "long" polling period, causing a host timeout to be violated

The more frequently an appication calls *USBMSC_poll()* during periods of peak mass storage activity, the higher the bandwidth will be. In contrast, having a long average period between *USBMSC_poll()* calls during heavy mass storage activity can result in very slow performance. When setting the average polling frequency, the developer should call *USBMSC_poll()* as often as the application can afford. It may be a good idea to perform experimental benchmarks.

The second concern is that one long period between *USBMSC_poll()* calls that could exceed a host timeout period. The timeout periods vary by operating system and situation. Most are fairly long, such that an occasional delay of even a few seconds may not cause the timeout to be exceeded. However, one notable exception is when the LUN is marked as having removable media (with *USBMSC_config*), on a Windows machine. Windows sends these LUNs a "TEST UNIT READY" SCSI command every second, to see if the medium is present. The MSC device has until the next TEST UNIT READY– that is, one second -- to respond. Some embedded applications could experience delays long enough to exceed this delay. If that were to happen, the Windows host would issue a USB bus reset; this should be avoided.

An advantage of using an RTOS is more direct control over the call frequency.

8.3.6 Processing Buffer Events

8.3.6.1 What Are They?

As discussed in Sec. 4.4.2, MSC applications typically require file system software. In this arrangement, both the application and the host (via the API) access the storage volume though the file system.

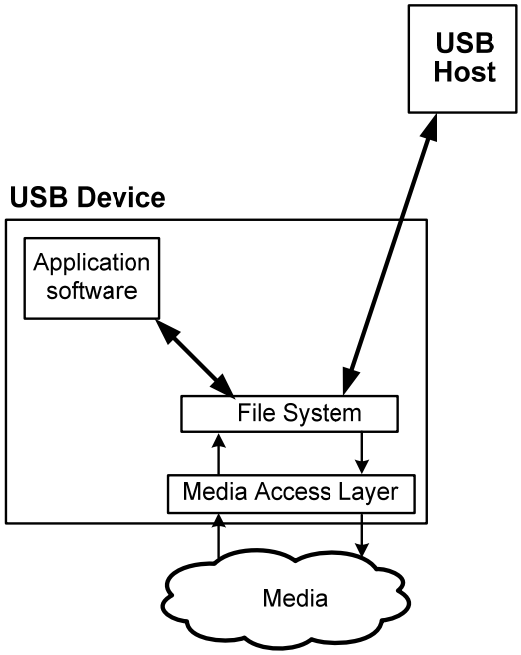


Figure 13. File System Access – Simplified

There is a wide variety of file system middleware on the market, with various specialties. For example, some are optimized for code size, while some have advanced features. No one set of file system software fits all applications. Therefore, a file system has not been integrated into the API. Rather, it exists at the application level, under the control of the software developer, allowing flexibility to choose the right one for the application. Putting the file system in the application requires a defined process by which the API can request the application to access the file system, which it needs to do during the handling of READ/WRITE SCSI commands from the host.

READ/WRITE operations usually involve multiple blocks per command, and each block (for the FAT file system) is typically 512 bytes. As a result, the total amount of data being moved for a single READ or WRITE operation is often fairly large. In many cases the data storage is not even within the MSP430 memory map, but rather in an off-chip medium. The combination of large data size, off-chip location, and limited RAM resources necessitates a multi-stage, iterative system in which data is paged between the medium and host through an intermediary RAM buffer.

This entire process is coordinated by the API; all the application must do is service *buffer operations* (that is, *process* the buffer) when requested to do so by the API. As the API prepares to send or receive a block as part of a multi-block READ/WRITE operation, it makes these requests:

- When handling READ commands, the application is requested to "fill" the buffer (i.e., using the file system to pull the data from the medium) so that the API can send it to the host.
- When handling WRITE commands, the application is requested to "empty" the buffer (i.e., using the file system to move the data into the medium) so that the API can receive more blocks from the host.

Since the API must ask the application to handle this function, the figure above isn't sufficient to describe what actually happens. Instead, see the figure below.

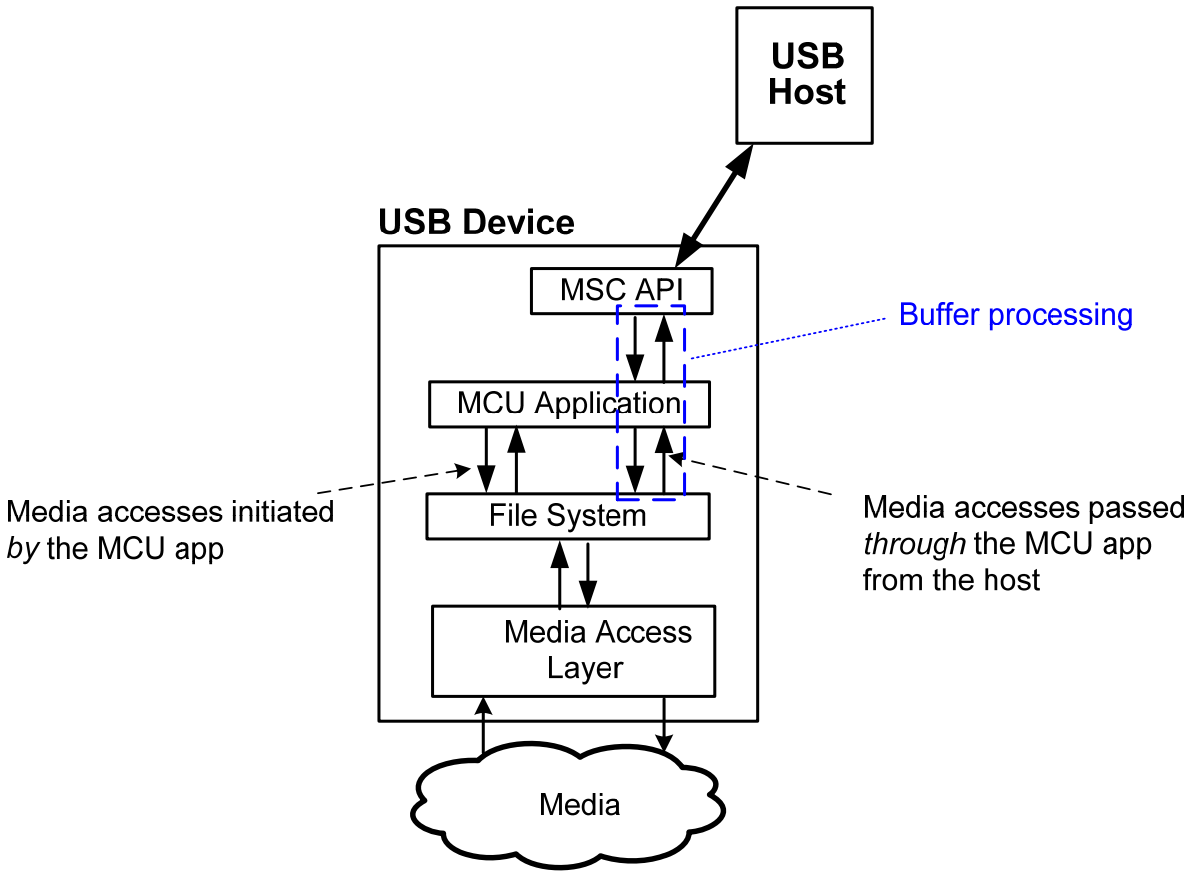


Figure 14. File System Access – Actual

The MCU application can access the media for its own purposes. It also acts as a go-between whenever the API needs media access to fulfill a READ/WRITE command from the host. The figure below details the latter, showing the complete cycle of a buffer operation.

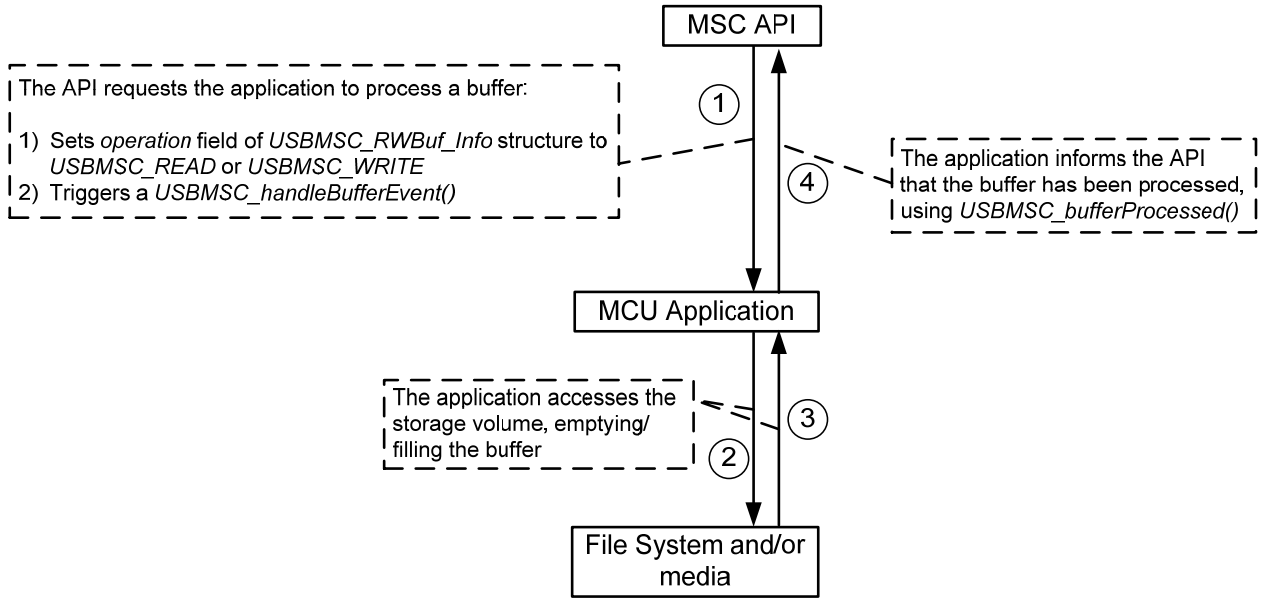


Figure 15. Buffer Processing

Buffer processing is further detailed below.

8.3.6.2 The *USBMSC_RWbuf_Info* Structure

The API defines an instance of this structure. (If the API is configured for double-buffering, the API defines two such structures.) It is considered a shared resource between the API and the application. Its purpose is to describe buffer operations requested by the API. Prior to USB enumeration, the application must call *USBMSC_fetchInfoStruct()* to obtain the pointer to this structure.

```
typedef struct
{
    BYTE         lun;
    BYTE         operation;
    DWORD        lba;
    BYTE         lbCount;
    BYTE         *bufferAddr;
    BYTE         returnCode;
    BYTE         XorY;
}USBMSC_RWbuf_Info;
```

Table 9. USBMSC_RWbuf_Info Definition

Type	Name	Direction	Description
BYTE	lun	From API to application	The logical unit on which the buffer operation is taking place. Zero-based.
BYTE	operation		The type of operation being performed (kUSBMSC_READ or kUSBMSC_WRITE), or NULL if no operation is active for this instance.
DWORD	lba		The logical block address (LBA) of the block the application needs to read/write.
BYTE	lbCount		The number of blocks being requested.
BYTE*	bufferAddr		The address of the RAM intermediary buffer the application should use to exchange the data.
BYTE	returnCode	From application to API	The result of the buffer operation. (To be written by the application.) Valid return codes are described in the definition of *USBMSC_bufferProcessed().*

8.3.6.3 The Lifecycle of a Buffer Operation

When the API wants to request the application to process a buffer:

1. It populates this structure. (As part of this, it sets the *operation* field to *kUSBMSC_READ* or *kUSBMSC_WRITE.*)
2. It clears the LPM bits in the MCU's status register, to ensure the CPU stays awake after the USB interrupt service routine exits. (These actions usually take place out of this ISR.)
3. It generates a *USBMSC_handleBufferEvent().*

To determine whether the API is waiting for the application to process a buffer, the application can check the *operation* field. A good place for this is immediately after calling *USBMSC_poll()* – refer to Sec. 8.3.5 regarding when in the application to do this. Alternatively, checking the *operation* field can be prompted from within *USBMSC_handleBufferEvent().*

Once the condition has been detected, the application should promptly process the buffer. It usually does this by accessing the medium to either fill the buffer (for READ operations) or empty it (for WRITE operations). It uses the information in the *USBMSC_RWbuf* instance to learn what kind of operation is being requested, and how it should be fulfilled. During this time, the host is waiting for the USB device to either send a block of data (READ) or to send status that the block has been written (WRITE).

When the media access is complete, the application must write a value into the *returnCode* field. Finally, it must call *USBMSC_bufferProcessed()* to inform the API that it has finished processing the buffer. This marks the end of the buffer operation lifecycle.

The API then resumes communication with the host. When it does, it uses the value in *returnCode* to tell the host the result of the operation. If an error was returned, the host will probably end the READ/WRITE SCSI command cycle. The information may further be passed to the application running on the host, in the form of an error code. The origin of this return code depends on the storage medium. If a file system is used, it is likely derived from error codes returned from the file system's function calls. Values the application may write into this field are shown in the table below.

Table 10. Accepted Values for *returnCode*

Name	Description
kUSBMSC_RWSuccess	The operation succeeded.
kUSBMSC_RWNotReady	The device was not ready.
kUSBMSC_RWIllegalReq	Illegal request
kUSBMSC_RWLbaOutOfRange	The LBA was out of range.
kUSBMSC_RWMedNotPresent	The media isn't present.
kUSBMSC_RWDevWriteFault	Device write fault
kUSBMSC_RWUnrecoveredRead	Unrecovered read
kUSBMSC_RWWriteProtected	The media is write protected

The buffer lifecycle is shown in the figure below.

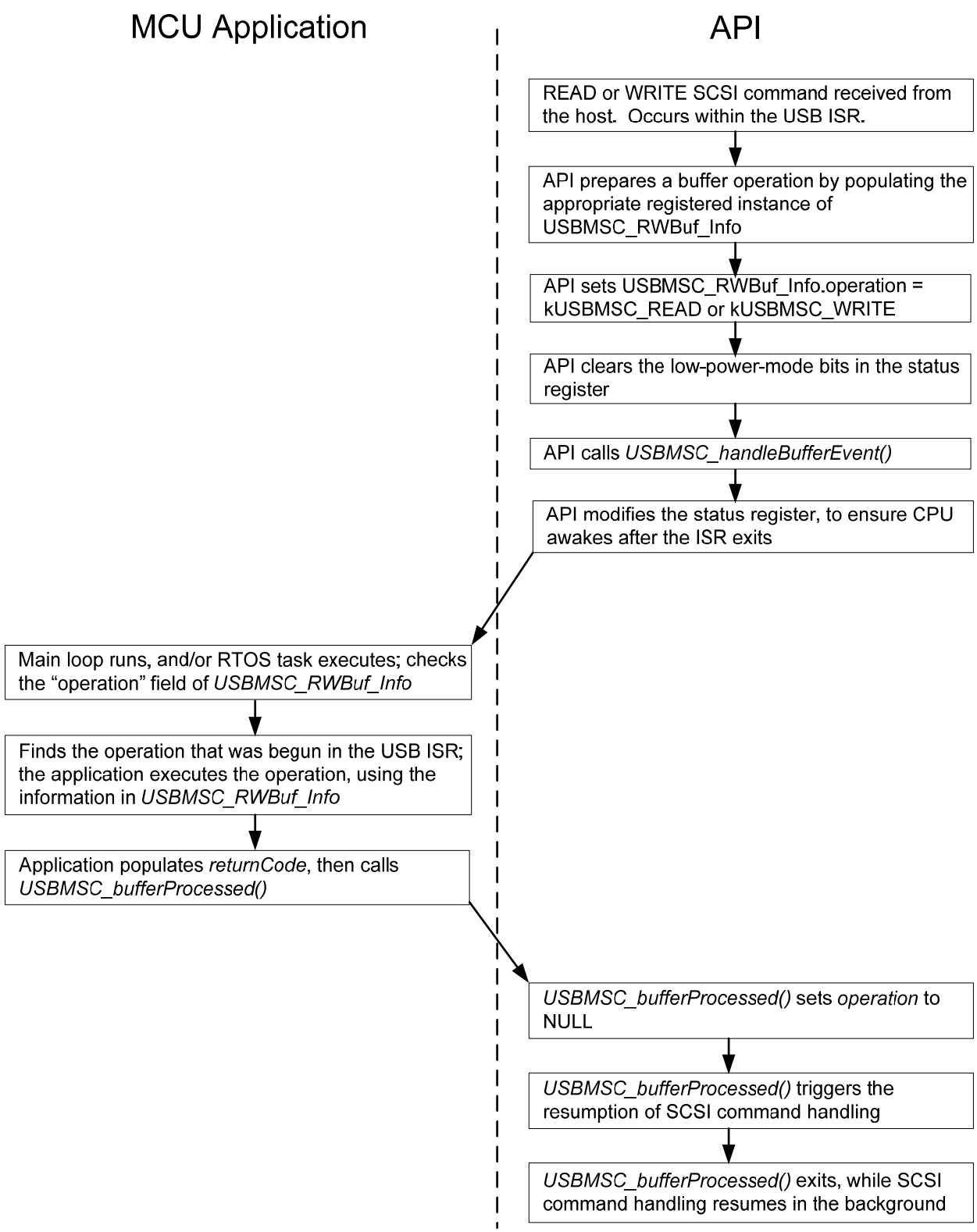


Figure 16. Buffer Lifecycle

8.4 Managing Dual Access to the Medium

The application should take care not to allow both itself and the USB host to access the medium at the same time. The host oftens keeps a cached version of the volume in its own memory that becomes out of sync with the one on the USB device. For this reason, it's best to avoid accessing the volume from the MCU application while the device is connected to a host as an MSC device.

9 Traditional HID Interfaces

This section applies to traditional HID interfaces. It does not apply to CDC interfaces, or to HID interfaces using the datapipe function calls.

9.1 Introduction

Traditional HID interfaces differ from those implemented by HID-datapipe in two respects:

- There is a need to use a report format that's different from the one used by the one used by HID-Datapipe. Thus the engineer must modify the default report descriptor in *descriptors.c.*
- The application makes calls to the HID-Traditional API function calls, rather than HID-Datapipe function calls. These calls access the interface at the report level, rather than at the higher level of sending/receives larger chunks of data.

HID report descriptors are essentially a scripting language by which the device imposes structure on the data at a low-level, rather than the unformatted datastream created by a COM port or HID-Datapipe device.

Generally speaking, there are only two situations in which HID-Traditional is advised rather than the HID-Datapipe:

- If the engineer wants to maintain backward-compatibility with an existing host application (i.e., Windows/Mac/Linux). In this case, the host application expects a particular format, and the new device must use the same.
- Applications in which the host OS itself acts as the "application". Examples of this are mice, keyboards, and tablets. In this case, the structure imposed by HID reports is necessary for the host and device to speak a common "language".

In the above cases, the engineer has no choice but to create an HID-Traditional device. In other situations, the heavy-handed, low-level structure imposed by reports typically doesn't add much value. The engineer typically controls both the device and the host applications anyway, and can impose structure of their own choosing at the application level. This makes a simple, unformatted datastream (like a COM port or HID-Datapipe) easier and faster than learning the complex structuring of HID reports.

This section describes the two cases above in more detail, and also outlines a general approach for others who wish to create a HID-Traditional "from scratch".

9.2 Overview of Creating a HID-Traditional Device

In all HID-Traditional cases, the recommended approach is:

1. Use the Descriptor Tool to generate a HID interface (which is actually a HID-Datapipe interface). Be sure to choose the correct polling interval (in the "interface view" tab) required for your application.
2. Modify *descriptors.c/h* with the custom report information.

3. In the application, use the HID-Traditional function calls to access the interface, rather than the the HID-Datapipe function calls

4. These calls require the application to build reports before sending, and parsing them after receiving

Step #1 is covered in Sec. 5. Detail on the remaining steps is provided below.

9.3 Obtaining the New Report Format

By definition, a traditional HID application requires a customized report format, achieved by customizing the default *report descriptor* (the one used by HID-Datapipe).

The first subsection reviews the default format, and the remaining ones describe common HID-Traditional situations, and where the format can be obtained in each case.

9.3.1 Review of the Default (HID-Datapipe) Report Format

The HID-Datapipe function calls require that the *default* report descriptor be used. This is the one created in *descriptors.c* by the Descriptor Tool:

```
BYTE const abromReportDescriptor[SIZEOF_REPORT_DESCRIPTOR]=
{
    0x06, 0x00, 0xff,  // Usage Page (Vendor Defined)
    0x09, 0x01,        // Usage Page (Vendor Defined)
    0xa1, 0x01,        // COLLECTION (Application)
    0x85, 0x3f,        // Report ID (Vendor Defined)
    0x95, MAX_PACKET_SIZE-1,      // Report Count
    0x75, 0x08,        // Report Size
    0x25, 0x01,        // Usage Maximum
    0x15, 0x01,        // Usage Minimum
    0x09, 0x01,        // Vendor Usage
    0x81, 0x02,        // Input (Data,Var,Abs)
    0x85, 0x3f,        // Report ID (Vendor Defined)
    0x95, MAX_PACKET_SIZE-1,      // Report Count
    0x75, 0x08,        // Report Size
    0x25, 0x01,        // Usage Maximum
    0x15, 0x01,        // Usage Minimum
    0x09, 0x01,        // Vendor Usage
    0x91 ,0x02,        // Input (Data,Var,Abs)
    0xc0               // end Application Collection
};
```

This describes the following report format, which was shown in Sec. 7.

Table 11. Reports Described by the Default Report Descriptor

Field	Size	Description
IN report (into the host)		
Report ID	1 byte	The report ID of the chosen report (automatically assigned to 0x3F by the HID-Datapipe calls)
Size	1 byte	The number of valid bytes in the *data* field
Data	62 bytes	Data payload
OUT report (out of the host)		
Report ID	1 byte	The report ID of the chosen report (must be assigned to 0x3F by the host)
Size	1 byte	The number of valid bytes in the *data* field
Data	62 bytes	Data payload

9.3.2 Creating a New Report for a Traditional HID where the Host OS is the "Application"

Devices like mice, keyboards, and tablets are unique, in that the host operating system itself acts as the "application". This is a major reason HID (and USB) was originally created. The engineer designing a mouse must format the data in a way the host OS will understand. The HID report descriptor "scripting language" exists to provide a semi-flexible way for the USB device to communicate to the host OS how its data is arranged. The host can then parse this descriptor to understand this formatting.

Those controlling host operating systems (Microsoft, Apple, and the Linux community) may further reduce the formatting options, because it reduces their effort to implement a host operating system that doesn't need to anticipate every possibility of report format.

These organizations provide example report formats to implement, for example, a keyboard, mouse, tablet, joystick, etc. Often these formats can be copied and pasted directly into *descriptors.c,* into the descriptor shown above in Sec. 9.3.1. (Some minor re-formatting of the syntax may be required.)

When changing the contents of the report descriptor array, it's necessary to update the value SIZEOF_REPORT_DESCRIPTOR that dimensions it. It's also necessary to update the value USBHID_REPORT_LENGTH, which is the length of the report described by the descriptor. The latter may be difficult to determine unless you understand HID report formatting; so hopefully the host OS documentation indicates this value as well.

As mentioned in Sec. 9.2, be sure to set the polling interval to the required value when using the Descriptor Tool to generate the HID interface. This value may be defined or recommended by the host OS documentation.

Once these are done, the API should be fully configured for the desired HID device. Code will need to be written at the application level that builds and parses the reports defined by the report descriptor.

9.3.3 Backward-Compatibility with a Previous HID Application

Some USB engineers use HID for general-purpose applications, developing both USB HID hardware and applications running on host platforms. Later, other engineers may need to design HID devices that are compatible with those host applications. These engineers must use whatever report format the previous engineers used.

If the later engineers are within the same company as the previous ones, it's likely the engineer already has access to source code from the previous USB device implementation that interacted with this host application. This would include the previous HID report descriptor. It can simply be copied and pasted into *descriptors.c,* replacing the default shown above in Sec. 9.3.1. (Some minor re-formatting of the syntax may be required.)

When changing the contents of the report descriptor array, it's necessary to update the value SIZEOF_REPORT_DESCRIPTOR that dimensions it. It's also necessary to update the value USBHID_REPORT_LENGTH, which is the length of the report described by the descriptor. Both values ought to be available within the previous device's source code.

As mentioned in Sec. 9.2, be sure to set the polling interval to the required value when using the Descriptor Tool to generate the HID interface. This value can be obtained from the previous device's source code.

If the engineer doesn't have access to the previous USB device's source code, another option is to use the source code for the host application. It may contain information useful in re-creating the HID report format. If this source isn't available either, a bus analyzer could be placed on the bus between the host and device as the device enumerates. This should be able to capture the report descriptor as it passes on the bus, allowing byte-by-byte reconstruction of the format.

Once these are done, the API should be fully configured for the desired HID device. Code will need to be written at the application level that builds the report defined by the report descriptor.

9.3.4 Creating a New Custom Report for Traditional HID Devices

If creating a new device, then a new HID report descriptor will need to be created. As mentioned previously, aside from the two cases described above, there's usually not much advantage to this approach, compared with using HID-Datapipe and imposing the needed structure (protocol, data formatting, etc.) at the application level. If this is what the engineer wants to do, however, the general procedure below can be used.

The USB Implementers Forum (USB-IF) provides a tool that can be used to create HID report descriptors, called the *HID Descriptor Tool.* (Note that despite the similar name, the functionality of the USB-IF's tool is very different than the *MSP430 HID Descriptor Tool*.)

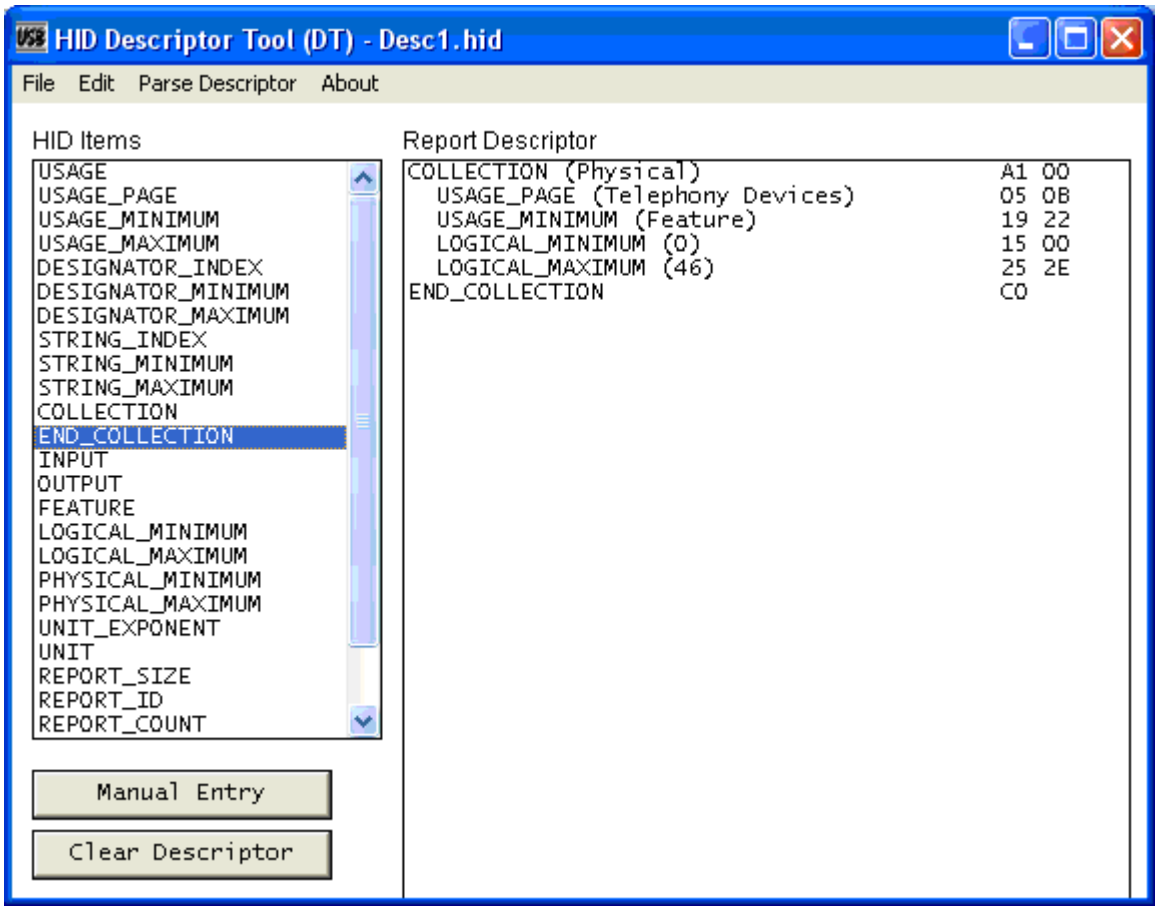


Figure 17. The USB-IF's *HID Descriptor Tool*

Describing use of this tool is outside the scope of this Programmer's Guide; the USB-IF and other sources describe it.

Once the report format is created, the output of this tool is a series of hex values. These values are very similar to the format of the default report descriptor shown in Sec. 9.3.1. Once the *MSP430 USB Descriptor Tool* is used to generate *descriptors.c/h* defining a device with a HID-Datapipe interface, the output of the USB-IF's *HID Descriptor Tool* can be pasted into the report descriptor in *descriptors.c/h.*

When changing the contents of the report descriptor array, it's necessary to update the value SIZEOF_REPORT_DESCRIPTOR that dimensions it. It's also necessary to update the value USBHID_REPORT_LENGTH, which is the length of the report described by the descriptor.

Once these are done, the API should be fully configured for the desired HID device. Code will need to be written at the application level that builds and parses the reports defined by the report descriptor.

9.4 Accessing the HID-Traditional Interface from the Application

The API provides functions for sending/receiving HID reports: *USBHID_sendReport()* and *USBHID_receiveReport(). USBHID_sendReport()* passes in a buffer (up to 64 bytes) containing, byte for byte, the report to be sent to the host the next time the host polls for a report. It is up to the application to format these 64 bytes in the exact same way they were defined by the report descriptor.

Similarly, *USBHID_receiveReport()* receives a buffer (up to 64 bytes) that is formatted in the manner described by the report descriptor. It can be called in response to a *USBHID_handleDataReceived()* event, or it can be polled at the polling interval that was chosen when generating *descriptors.c* with the Descriptor Tool.

(Note that although reports themselves may be of any size up to 64 bytes (defined by USBHID_REPORT_LENGTH in *descriptors.h*), the actual USB packet size (MAX_PACKET_SIZE in *descriptors.h*) currently must be 64 bytes. Usually there isn't a downside to this, other than using up slightly more bandwidth than truly needed.)

The report descriptor might define multiple reports, defined by "report ID" fields. The application can therefore send/receive multiple reports by setting the correct report ID value prior to calling *USBHID_sendReport(),* or can separate reports once received with *USBHID_receiveReport()* by reading the report ID field.

All bytes described by the report descrirptor are the responsibility of the application; the API does not change them.

Currently the API is restricted to use 64-byte packets. A report can be defined that is less than this, but the packet size on the bus will still be 64 bytes.

10 API Function Calls

10.1 MSP430 USB Module Management

These calls configure and manage the MSP430's USB module. They are shared among all the API stack interface types (CDC/HID/MSC). They are all defined within the *usb.c* source file.

Table 12. USB Module Management Call Summary

Function	Description
`BYTE USB_init()`	Initializes the USB hardware interface by configuring interrupts, power, and clocks, but does not activate the PLL or transceiver.
`BYTE USB_enable()`	Activates the USB module, PLL, and transceiver.
`BYTE USB_disable()`	Disable USB module, PLL, and transceiver.
`BYTE USB_setEnabledEvents()`	Enables/disables various USB events
`BYTE USB_getEnabledEvents()`	Returns the status of USB event enabling

10.1.1 BYTE USB_init(void)

Description

Initializes the USB module by configuring power and clocks, and configures pins that are critical for USB. This should be called prior to any other API functions, usually at the beginning of program execution.

Note that this does not enable the USB module (that is, does not set USB_EN bit). Rather, it prepares the USB module to detect the application of power to VBUS, after which the application may choose to enable the module and connect to USB. As such, power consumption does not increase when this function is executed.

Parameters

Table 13. Parameters for *USB_init()*

returns	`kUSB_succeed`

10.1.2 BYTE USB_enable(void)

Description

Enables the USB module, which includes activating the PLL and setting the USB_EN bit. Power consumption increases as a result of this operation (see device datasheet for specifics). This call should only be made after an earlier call to *USB_init(),* and prior to any other call except than *USB_setEnabledEvents(),* or *USB_getEnabledEvents().* It is usually called just prior to attempting to connect with a host after a bus connection has already been detected.

Parameters

Table 14. Parameters for *USB_enable*()

Returns	kUSB_succeed

10.1.3 BYTE USB_disable(void)

Description

Disables the USB module and PLL. If USB is not enabled when this call is made, no error is returned – the call simply exits with success.

If a *handleVbusOffEvent()* occurs, or if *USB_connectionState()* begins returning ST_USB_DISCONNECTED, this function should be called (following a call to *USB_disconnect()*), in order to avoid unnecessary current draw.

Parameters

Table 15. Parameters for *USB_disable()*

Returns	kUSB_succeed

10.1.4 BYTE USB_setEnabledEvents(WORD events)

Description

Enables/disables various USB events. Within the *events* byte, all bits with '1' values will be enabled, and all bits with '0' values will be disabled. (There are no bit-wise operations). By default (that is, prior to any call to this function), all events are disabled.

The status of event enabling can be read with the *USB_getEnabledEvents()* function. This call can be made at any time after a call to *USB_init().*

USB_setEnabledEvents() can be thought of in a similar fashion to setting/clearing interrupt enable bits. The only benefit in keeping an event disabled is to save the unnecessary execution cycles incurred from running an "empty" event handler.

The mask constant *kUSB_allUsbEvents* is used to enable/disable all events pertaining to core USB functions; in other words, it enables all those with a *kUSB_* prefix.

See Sec. 11 for more information about events.

Parameters

Table 16. Parameters for *USB_setEnabledEvents*()

WORD events	passed in	kUSB_clockFaultEvent

		kUSB_VbusOnEvent kUSB_VbusOffEvent kUSB_UsbResetEvent kUSB_UsbSuspendEvent kUSB_UsbResumeEvent kUSBCDC_dataReceivedEvent kUSBCDC_sendCompletedEvent kUSBCDC_receiveCompletedEvent kUSBHID_dataReceivedEvent kUSBHID_sendCompletedEvent kUSBHID_receiveCompletedEvent kUSB_allUsbEvents
Returns		kUSB_succeed

10.1.5 WORD USB_getEnabledEvents(void)

Description

Returns which events are enabled and which are disabled. The definition of *events* is the same as for *USB_enableEvents()* above.

If the bit is set, the event is enabled. If cleared, the event is disabled. By default (that is, prior to calling *USB_setEnabledEvents()*), all events are disabled.

This call can be made at any time after a call to *USB_init().*

Parameters

Table 17. Parameters for *USB_getEnabledEvents*()

Returns	kUSB_clockFaultEvent kUSB_VbusOnEvent kUSB_VbusOffEvent kUSB_UsbResetEvent kUSB_UsbSuspendEvent kUSB_UsbResumeEvent kUSBCDC_dataReceivedEvent kUSBCDC_sendCompletedEvent kUSBCDC_receiveCompletedEvent

10.2 USB Connection Management

These calls pertain to the USB connection with the host. They are shared among all the API stack interface types (CDC/HID/MSC). They are defined within the *usb.c* source file.

Table 18. USB Connection Management Call Summary

Function	Description
`BYTE USB_reset()`	Resets the USB module and the internal state of the API.
`BYTE USB_connect()`	Instructs USB module to make itself available to the host for connection, by pulling the PUR pin high.
`BYTE USB_disconnect()`	Forces a disconnect from the host by pulling the PUR pin low
`BYTE USB_forceRemoteWakeup()`	Forces a remote wakeup of the USB host
`BYTE USB_connectionInfo()`	Returns low-level information about the USB connection
`BYTE USB_connectionState()`	Returns the state of the USB connection

10.2.1 BYTE USB_reset(void)

Description

Resets the USB module and also the internal state of the API. The interrupt register is cleared to make sure no interrupts are pending. If the device had been enumerated, the enumeration is now lost. All open send/receive operations are aborted.

This function is most often called immediately before a call to *USB_connect().* It should not be called prior to *USB_enable().*

Parameters

Table 19. Parameters for *USB_reset* ()

Returns	`kUSB_succeed`

10.2.2 BYTE USB_connect(void)

Description

Instructs the USB module to make itself available to the host for connection, by pulling the D+ signal high using the PUR pin. This call should only be made after a call to *USB_enable().*

Parameters

Table 20. Parameters for *USB_connect()*

Returns	kUSB_succeed

10.2.3 BYTE USB_disconnect(void)

Description

Forces a logical disconnect from the USB host by pulling the PUR pin low, removing the pullup on the D+ signal. The USB module and PLL remain enabled. If the USB is not connected when this call is made, no error is returned – the call simply exits with success after ensuring PUR is low.

Parameters

Table 21. Parameters for *USB_disconnect()*

Returns	kUSB_succeed

10.2.4 BYTE USB_forceRemoteWakeup(void)

Description

Prompts a remote wakeup of the USB host. The user must ensure that the USB descriptors had indicated remote wakeup capability (using the Descriptor Tool); otherwise the host will ignore the request.

If the function returns *kUSB_generalError*, it means that the host did not grant the device the ability to perform a remote wakeup, when it enumerated the device.

Parameters

Table 22. Parameters for *USB_forceRemoteWakeup()*

Returns	kUSB_succeed kUSB_generalError kUSB_notSuspended

10.2.5 BYTE USB_connectionState(void)

Description

Returns the state of the USB connection, according to the state diagram in Sec. 6.

Parameters

Table 23. Parameters for *USB_connectionState()*

Returns	ST_USB_DISCONNECTED ST_USB_CONNECTED_NO_ENUM ST_ENUM_IN_PROGRESS ST_ENUM_ACTIVE ST_ENUM_SUSPENDED ST_NOENUM_SUSPENDED ST_ERROR

10.2.6 BYTE USB_connectionInfo(void)

Description

Returns low-level status information about the USB connection.

Because multiple flags can be returned, the possible values can be masked together – for example, *kUSB_vbusPresent + kUSB_suspended.*

Parameters

Table 24. Parameters for *USB_connectionInfo()*

Returns	kUSB_vbusPresent kUSB_purHigh kUSB_suspended kUSB_NotSuspended kUSB_Enumerated

10.3 CDC Management and Data Handling

These calls are specific to the Communications Device Class. They are defined within the *usbcdc.c* source file.

Table 25. CDC Management and Data Handling Call Summary

Function	Description
`BYTE USBCDC_sendData()`	Begins a send operation to the USB host
`BYTE USBCDC_receiveData()`	Begins a receive operation from the USB host
`BYTE USBCDC_bytesInUSBBuffer()`	Returns the number of bytes residing in the USB endpoint buffer awaiting a receive operation to move them to a user buffer.
`BYTE USBCDC_abortSend()`	Aborts an active send operation
`BYTE USBCDC_abortReceive()`	Aborts an active receive operation
`BYTE USBCDC_rejectData()`	Rejects payload data residing in the USB buffer, for which a receive operation has not yet been initiated
`BYTE USBCDC_intfStatus()`	Returns status information specific to a particular CDC interface

10.3.1 BYTE USBCDC_sendData(BYTE * data, WORD size, BYTE intfNum)

Description

Initiates sending of a user buffer over CDC interface *intfNum*, of size *size* and starting at address *data.* If *size* is larger than the packet size, the function handles all packetization and buffer management. *size* has no inherent upper limit (beyond being a 16-bit value).

In most cases where a send operation is successfully started, the function will return *kUSBCDC_sendStarted.* A *send operation* is said to be underway. At some point, either before or after the function returns, the send operation will complete, barring any events that would preclude it. (Even if the operation completes before the function returns, the return code will still be *kUSBCDC_sendStarted.)*

If the bus is not connected when the function is called, the function returns *kUSBCDC_busNotAvailable,* and no operation is begun. If *size* is 0, the function returns *kUSBCDC_generalError.* If a previous send operation is already underway for this data interface, the function returns with *kUSBCDC_intfBusyError.*

USB includes low-level mechanisms that ensure valid transmission of data.

See Sec. 7.2 for a detailed discussion of send operations.

Parameters

Table 26. Parameters for *USBCDC_sendData()*

`BYTE* data`	passed in	An array of data to be sent
`WORD size`	passed in	Number of bytes to be sent, starting from address *data*.
`BYTE intfNum`	passed in	Which data interface the data should be transmitted over
Returns		`kUSBCDC_sendStarted`: a send operation was successfully started `kUSBCDC_intfBusyError`: a previous send operation is underway `kUSBCDC_busNotAvailable`: the bus is either suspended or disconnected `kUSBCDC_generalError`: size was zero, or other error

10.3.2 BYTE USBCDC_receiveData(BYTE * data, WORD size, BYTE intfNum)

Description

Receives *size* bytes over CDC interface *intfNum* into memory starting at address *data. size* has no inherent upper limit (beyond being a 16-bit value).

The function may return with *kUSBCDC_receiveStarted*, indicating that a receive operation is underway. The operation completes when *size* bytes are received. The application should ensure that the *data* memory buffer be available during the whole of the receive operation.

The function may also return with *kUSBCDC_receiveCompleted.* This means that the receive operation was complete by the time the function returned.

If the bus is not connected when the function is called, the function returns *kUSBCDC_busNotAvailable,* and no operation is begun. If *size* is 0, the function returns *kUSBCDC_generalError.* If a previous receive operation is already underway for this data interface, the function returns *kUSBCDC_intfBusyError.*

USB includes low-level mechanisms that ensure valid transmission of data.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 27. Parameters for *USBCDC_receiveData()*

`BYTE* data`	passed in	An array to contain the data received.
`WORD size`	passed in	Number of bytes to be received
`BYTE intfNum`	passed in	Which data interface to receive from
Returns		`kUSBCDC_receiveStarted`: A receive operation has been succesfully started. `kUSBCDC_receiveCompleted`: The receive operation is already completed. `kUSBCDC_intfBusyError`: a previous receive operation is underway `kUSBCDC_busNotAvailable`: the bus is either suspended or disconnected `kUSBCDC_generalError`: size was zero, or other error

10.3.3 BYTE USBCDC_bytesInUSBBuffer(BYTE intfNum)

Description

Returns the number of bytes waiting in the USB endpoint buffer for *intfNum.* A non-zero value generally means that no receive operation is open by which these bytes can be copied to a user buffer. If the value is non-zero, the application should either open a receive operation so that the data can be moved out of the endpoint buffer, or the data should be rejected (*USBCDC_rejectData()*).

Parameters

Table 28. Parameters for *USBCDC_bytesInUSBBuffer()*

`BYTE intfNum`	passed in	The data interface whose buffer is to be checked
Returns		The number of bytes waiting in this buffer

10.3.4 BYTE USBCDC_abortSend(WORD* size, BYTE intfNum)

Description

Aborts an active send operation on data interface *intfNum.* Returns the number of bytes that were sent prior to the abort, in *size.*

An application may choose to call this function if sending failed, due to factors such as:

- a surprise removal of the bus
- a USB suspend event
- any send operation that extends longer than desired (perhaps due to no open COM port on the host.)

Parameters

Table 29. Parameters for *USBCDC_abortSend()*

`WORD* size`	passed out	Number of bytes that were sent prior to the abort action.
`BYTE intfNum`	passed in	The data interface for which the send should be aborted
Returns		`kUSB_succeed`

10.3.5 BYTE USBCDC_abortReceive(WORD* size, BYTE intfNum)

Description

Aborts an active receive operation on CDC interface *intfNum.* Returns the number of bytes that were received and transferred to the data location established for this receive operation. The data moved to the buffer up to that time remains valid.

An application may choose to call this function if it decides it no longer wants to receive data from the USB host. It should be noted that if a continuous stream of data is being received from the host, aborting the operation is akin to pressing a “pause” button; the host will be NAK’ed until another receive operation is opened.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 30. Parameters for *USBCDC_abortReceive()*

`WORD* size`	passed out	Number of bytes that were received and are waiting at the assigned address.
`BYTE intfNum`	passed in	The data interface for which the send should be aborted
Returns		`kUSB_succeed`

10.3.6 BYTE USBCDC_rejectData(BYTE intfNum)

Description

This function rejects data that has been received from the host, for interface *inftNum,* that does not have an active receive operation underway. It resides in the USB endpoint buffer and blocks further data until a receive operation is opened, or until rejected. When this function is called, the buffer for this interface is purged, and the data lost. This frees the USB path to resume communication.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 31. Parameters for *USBCDC_rejectData()*

Returns	kUSB_succeed

10.3.7 BYTE USBCDC_intfStatus(BYTE intfNum, WORD* bytesSent, WORD* bytesReceived)

Description

Indicates the status of the CDC interface *intfNum*. If a send operation is active for this interface, the function also returns the number of bytes that have been transmitted to the host. If a receive operation is active for this interface, the function also returns the number of bytes that have been received from the host and are waiting at the assigned address.

Because multiple flags can be returned, the possible values can be masked together – for example, *kUSBCDC_waitingForSend + kUSBCDC_dataWaiting.*

Parameters

Table 32. Parameters for *USBCDC_intfStatus()*

`BYTE intfNum`	passed in	Interface number for which status is being retrieved.
`WORD* bytesSent`	passed out	If a send operation is underway, the number of bytes that have been transferred to the host is returned in this location. If no send operation is underway, this returns zero.
`WORD* bytesReceived`	passed out	If a receive operation is underway, the number of bytes that have been transferred to the assigned memory location is returned in this location. If no receive operation is underway, this returns zero.
Returns		`kUSBCDC_waitingForSend`: Indicates that a send operation is open on this interface `kUSBCDC_waitingForReceive`: Indicates that a receive operation is open on this interface `kUSBCDC_dataWaiting`: Indicates that data has been received from the host for this interface, waiting in the USB receive buffers, lacking an open receive operation to accept it. `kUSBCDC_busNotAvailable`: Indicates that the bus is either suspended or disconnected. Any operations that had previously been underway are now aborted.

10.4 HID-Datapipe: Management and Data Handling

These calls are specific to HID interfaces. They are used to implement a datapipe, as opposed to a traditional HID device, as discussed in Sec. 4.3.

These calls will only work properly using the default report format output by the Descriptor Tool. They should not be mixed with the traditional HID calls on a given interface. The exceptions to this are *USBHID_intfStatus()* and *USBHID_rejectData().*

They are defined within the HID source files listed in Sec. 4.1.5.

Table 33. HID-Datapipe Management and Data Handling Call Summary

Function	Description
`BYTE USBHID_sendData()`	Begins a send operation to the USB host
`BYTE USBHID_receiveData()`	Begins a receive operation from the USB host
`BYTE USBHID_bytesInUSBBuffer()`	Returns the number of bytes from the host waiting in the USB endpoint buffer
`BYTE USBHID_abortSend()`	Aborts an active send operation
`BYTE USBHID_abortReceive()`	Aborts an active receive operation
`BYTE USBHID_rejectData()`	Rejects payload data residing in the USB buffer, for which a receive operation has not yet been initiated
`BYTE USBHID_intfStatus()`	Returns status information specific to a particular HID interface

*10.4.1 BYTE USBHID_sendData(BYTE * data, WORD size, BYTE intfNum)*

Description

Initiates sending of a user buffer over HID interface *intfNum*, of size *size* and starting at address *data.* If *size* is larger than the packet size, the function handles all packetization and buffer management. *size* has no inherent upper limit (beyond being a 16-bit value).

In most cases where a send operation is successfully started, the function will return *kUSBHID_sendStarted.* A *send operation* is said to be underway. At some point, either before or after the function returns, the send operation will complete, barring any events that would preclude it. (Even if the operation completes before the function returns, the return code will still be *kUSBHID_sendStarted.)*

If the bus is not connected when the function is called, the function returns *kUSBHID_busNotAvailable,* and no operation is begun. If *size* is 0, the function returns *kUSBHID_generalError.* If a previous send operation is already underway for this data interface, the function returns with *kUSBHID_intfBusyError.*

USB includes low-level mechanisms that ensure valid transmission of data.

See Sec. 7.2 for a detailed discussion of send operations.

Parameters

Table 34. Parameters for *USBHID_sendData()*

`BYTE* data`	passed in	An array of data to be sent
`WORD size`	passed in	Number of bytes to be sent, starting from address *data*.
`BYTE intfNum`	passed in	Which data interface the data should be transmitted over
Returns		`kUSBHID_sendStarted`: a send operation was successfully started `kUSBHID_intfBusyError`: a previous send operation is underway `kUSBHID_busNotAvailable`: the bus is either suspended or disconnected `kUSBHID_generalError`: size was zero, or other error

*10.4.2 BYTE USBHID_receiveData(BYTE * data, WORD size, BYTE intfNum)*

Description

Receives *size* bytes over HID interface *intfNum* into memory starting at address *data. size* has no inherent upper limit (beyond being a 16-bit value).

The function may return with *kUSBHID_receiveStarted*, indicating that a receive operation is underway. The operation completes when *size* bytes are received. The application should ensure that the *data* memory buffer be available during the whole of the receive operation.

The function may also return with *kUSBHID_receiveCompleted.* This means that the receive operation was complete by the time the function returned.

If the bus is not connected when the function is called, the function returns *kUSBHID_busNotAvailable,* and no operation is begun. If *size* is 0, the function returns *kUSBHID_generalError.* If a previous receive operation is already underway for this data interface, the function returns *kUSBHID_intfBusyError.*

USB includes low-level mechanisms that ensure valid transmission of data.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 35. Parameters for *USBHID_receiveData()*

`BYTE* data`	passed in	An array to contain the data received.
`WORD size`	passed in	Number of bytes to be received
`BYTE intfNum`	passed in	Which data interface to receive from
Returns		`kUSBHID_receiveStarted`: A receive operation has been successfully started. `kUSBHID_receiveCompleted`: The receive operation is already completed. `kUSBHID_intfBusyError`: a previous receive operation is underway `kUSBHID_ busNotAvailable`: the bus is either suspended or disconnected `kUSBHID_generalError`: size was zero, or other error

10.4.3 BYTE USBHID_bytesInUSBBuffer(BYTE intfNum)

Description

Returns the number of bytes waiting in the USB endpoint buffer for *intfNum.* A non-zero value generally means that no receive operation is open by which these bytes can be copied to a user buffer. If the value is non-zero, the application should either open a receive operation so that the data can be moved out of the endpoint buffer, or the data should be rejected (*USBHID_rejectData()*).

Parameters

Table 36. Parameters for *USBHID_bytesInUSBBuffer()*

`BYTE intfNum`	passed in	The data interface whose buffer is to be checked
Returns		The number of bytes waiting in this buffer

10.4.4 BYTE USBHID_abortSend(WORD size, BYTE intfNum)*

Description

Aborts an active send operation on data interface *intfNum.* Returns the number of bytes that were sent prior to the abort, in *size.*

An application may choose to call this function if sending failed, due to factors such as:

- a surprise removal of the bus
- a USB suspend event
- any send operation that extends longer than desired

Parameters

Table 37. Parameters for *USBHID_abortSend()*

`WORD* size`	passed out	Number of bytes that were sent prior to the abort action.
`BYTE intfNum`	passed in	The data interface for which the send should be aborted
Returns		`kUSB_succeed`

10.4.5 BYTE USBHID_abortReceive(WORD* size, BYTE intfNum)

Description

Aborts an active receive operation on HID interface *intfNum.* Returns the number of bytes that were received and transferred to the data location established for this receive operation. The data moved to the buffer up to that time remains valid.

An application may choose to call this function if it decides it no longer wants to receive data from the USB host. It should be noted that if a continuous stream of data is being received from the host, aborting the operation is akin to pressing a “pause” button; the host will be NAK’ed until another receive operation is opened.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 38. Parameters for *USBHID_abortReceive()*

`WORD* size`	passed out	Number of bytes that were received and are waiting at the assigned address.
`BYTE intfNum`	passed in	The data interface for which the receive should be aborted
Returns		`kUSB_succeed`

10.4.6 BYTE USBHID_rejectData(BYTE intfNum)

Description

This function rejects data that has been received from the host, for interface *intfNum,* that does not have an active receive operation underway. It resides in the USB endpoint buffer and blocks further data until a receive operation is opened, or until rejected. When this function is called, the buffer for this interface is purged, and the data lost. This frees the USB path to resume communication.

See Sec. 7.2 for a detailed discussion of receive operations.

Parameters

Table 39. Parameters for *USBHID_rejectData()*

Returns	`kUSB_succeed`

10.4.7 BYTE USBHID_intfStatus(BYTE intfNum, WORD bytesSent, WORD* bytesReceived)*

Description

Indicates the status of the HID interface *intfNum*. If a send operation is active for this interface, the function also returns the number of bytes that have been transmitted to the host. If a receive operation is active for this interface, the function also returns the number of bytes that have been received from the host and are waiting at the assigned address.

Because multiple flags can be returned, the possible values can be masked together – for example, *kUSBHID_waitingForSend + kUSBHID_dataWaiting.*

Parameters

Table 40. Parameters for *USBHID_intfStatus()*

`BYTE intfNum`	passed in	Interface number for which status is being retrieved.
`WORD* bytesSent`	passed out	If a send operation is underway, the number of bytes that have been transferred to the host is returned in this location. If no send operation is underway, this returns zero.
`WORD* bytesReceived`	passed out	If a receive operation is underway, the number of bytes that have been transferred to the assigned memory location is returned in this location. If no receive operation is underway, this returns zero.
Returns		`kUSBHID_waitingForSend`: Indicates that a send operation is open on this interface `kUSBHID_waitingForReceive`: Indicates that a receive operation is open on this interface `kUSBHID_dataWaiting`: Indicates that data has been received from the host for this interface, waiting in the USB receive buffers, lacking an open receive operation to accept it. `kUSBHID_busNotAvailable`: Indicates that the bus is either suspended or disconnected. Any operations that had previously been underway are now aborted.

10.5 HID-Traditional: Management and Data Handling

These calls are specific to HID interfaces. They are used to implement a traditional HID device, as opposed to a datapipe, as discussed in Sec. 4.3.

These calls should not be mixed with HID-Datapipe calls on a given interface. The exceptions to this are *USBHID_intfStatus()* and *USBHID_rejectData(),* which are defined in the previous section.

They are defined within the HID source files listed in Sec. 4.1.5.

Table 41. HID-Traditional Management and Data Handling Call Summary

Function	Description
`BYTE USBHID_sendReport()`	Sends a report to the USB host
`BYTE USBHID_receiveReport()`	Receives a report from the USB host

*10.5.1 BYTE USBHID_sendReport(BYTE * reportData, BYTE intfNum)*

Sends a pre-built report *reportData* to the host, on interface *intfNum*. The report must be organized to reflect the format defined by the report descriptor in *descriptors.c*.

When the function returns *kUSBHID_sendComplete*, the data has been written to the USB transmit buffers, and will be transferred to the host in the next polling frame. If the function returns *kUSBHID_busNotAvailable,* then the bus has either been disconnected or the device is suspended, allowing no reports to be sent. If the function returns *kUSBHID_intfBusyError,* it means the USB buffer for the interface has data in it, suggesting that the host has not yet fetched the previously-loaded report.

Parameters

Table 42. Parameters for *USBHID_sendReport()*

`BYTE* reportData`	passed in	An array containing the report
`BYTE intfNum`	passed in	Which HID interface the data should be transmitted over

Returns		`kUSBHID_sendComplete`
		`kUSBHID_busNotAvailable`
		`kUSBHID_intfBusyError`

*10.5.2 BYTE USBHID_receiveReport(BYTE * reportData, BYTE intfNum)*

Receives a report from the host into *reportData*, on interface *intfNum*. It is expected that the host will organize the report in the format defined by the report descriptor in *descriptors.c*.

When the function returns *kUSBHID_receiveCompleted*, the data has been successfully copied from the USB receive buffers into *reportData*. If the function returns *kUSBHID_busNotAvailable,* then the bus has either been disconnected or the device is suspended, allowing no reports to be sent. If the function returns *kUSBHID_generalError,* it means the call failed for unspecified reasons.

The call is intended to be called only when it is known that a report is in the USB buffer. This means it is best called in response to the API calling *USBHID_handleDataReceived(),* which indicates that a report has been received for interface *intfNum.*

Parameters

Table 43. Parameters for *USBHID_receiveReport()*

`BYTE* reportData`	passed in	An array containing the report
`BYTE intfNum`	passed in	The HID interface over which the data is to be received

Returns		`kUSBHID_busNotAvailable`
		`kUSBHID_receiveCompleted`
		`kUSBHID_generalError`

10.6 MSC: Management and Data Handling

These calls are specific to MSC interfaces. They are defined within the MSC source files listed in Sec. 4.1.5.

Table 44. MSC Management and Data Handling Call Summary

Function	Description
BYTE USBMSC_poll()	Initiates the handling of the current SCSI command for this MSC interface.
BYTE USBMSC_registerBufInfo()	Registers with the API which buffers it should use for READ/WRITE command handling.
BYTE USBMSC_fetchInfoStruct()	Returns the pointer(s) to the API's instance(s) of *USBMSC_RWBuf_Info*
BYTE USBMSC_bufferProcessed()	Returns a buffer back to the API after processing it, in the course of SCSI READ or WRITE handling.
BYTE USBMSC_updateMediaInfo()	Posts an update to the host about the media being used for a LUN with removable media.

10.6.1 BYTE USBMSC_poll()

Description

Checks to see if a SCSI command has been received. If so, it handles it. If not, it returns having taken no action.

The return values of this function are intended to be used with entry of low-power modes. If the function returns *kUSBMSC_okToSleep,* then no further application action is required; that is, either no SCSI command was received; one was received but immediately handled; or one was received but the handling will be completed in the background by the API as it automatically services USB interrupts.

If instead the function returns *kUSBMSC_processBuffer,* then the API is currently servicing a SCSI READ or WRITE command, and the API requires the application to process a buffer. (See Sec. 8.3.6 for a discussion of buffer processing.)

Note that even if the function returns these values, the values could potentially be outdated by the time the application evaluates them. For this reason, it's important to disable interrupts prior to calling this function. See Sec. 8.3.5 for more information.

Parameters

Table 45. Parameters for *USBMSC_poll()*

Returns	*kUSBMSC_okToSleep* *kUSBMSC_processBuffer*

10.6.2 BYTE USBMSC_registerBufInfo(BYTE xBufAddr, BYTE* yBufAddr, WORD size)*

Description

Gives the API a buffer to use for READ/WRITE data transfer. *size* indicates the size of the buffer, in bytes.

Note: Currently, only single-buffering is supported, so *yBufAddr* should be set to null.

If the application intends to allocate the buffer statically, then this function needs only to be called once, prior to any READ/WRITE commands being received from the host. Most likely this would happen during the application's initialization functions.

However, this function optionally enables dynamic buffer management. That is, it can activate and de-activate the buffer, by alternately assigning a null and valid address in *xBufAddr*. This is useful because the buffer uses a significant portion of the RAM resources (typically 512 bytes). This memory is not needed when USB is not attached or suspended.

If doing this, it's important that the application re-activate the buffer when USB becomes active again, by issuing another call to the function, this time using valid buffer information. If the API needs the buffer and doesn't have it, it will begin failing READ/WRITE commands from the host. The re-activation can take place within *handleVbusOffEvent()*.

size must be a multiple of a block size – for FAT, a block size is typically 512 bytes. Thus values of 512, 1024, 1536, etc. are valid. Non-multiples are not valid.

The function returns *kUSB_succeed* every time. It is up to the application to ensure that the buffers are valid.

Parameters

Table 46. Parameters for *USBMSC_registerBufInfo()*

BYTE* *xBufAddr*	The address of an X-buffer. If null, then both buffers are de-activated.
BYTE * *yBufAddr*	The address of an Y-buffer. (Double-buffering is not supported in this version of the API.)
WORD *size*	The size, in bytes, of the buffers.
Returns	kUSB_succeed

10.6.3 USBMSC_RWbuf_Info* USBMSC_fetchInfoStruct(VOID)

Description

Returns a pointer to the *USBMSC_RWbuf_Info* structure instance maintained within the API. See Sec. 8.3.6 for information on using this structure.

This function should be called prior to USB enumeration; that is, prior to calling *USB_connect().*

Parameters

Table 47. Parameters for *USBMSC_fetchInfoStruct()*

Returns	A pointer to an application-allocated instance of *USBMSC_RWBuf_Info*, which will be used to exchange information related to buffer requests from the API to the application.

10.6.4 BYTE USBMSC_bufferProcessed(USBMSC_RWbuf_Info * RWBufInfo)

Description

This function should be called by the application after it has processed a buffer request. It indicates to the API that the application has fulfilled the request.

Prior to calling this function, the application needs to write a return code to *rwInfo.returnCode.* This code should reflect the result of the operation. The value may come from the file system software, depending on the application. See Sec. 8.3.6 for a list of valid return codes.

Parameters

Table 48. Parameters for *USBMSC_bufferProcessed()*

USBMSC_RWbuf_Info * *RWBufInfo*	Pass the value received from *USBMSC_fetchInfoStruct()*
Returns	*kUSBMSC_succeed*

10.6.5 BYTE USBMSC_updateMediaInfo(BYTE lun, USBMSC_mediaInfoStr * info)

Description

Informs the API of the current state of the media on LUN *lun.* It does this using an instance *info* of the API-defined structure *USBMSC_mediaInfoStr.* The API uses the information in the most recent call to this function in automatically handling certain requests from the host.

In LUNs that are marked as not removable in *USBMSC_CONFIG,* this function should be called once at the beginning of execution, prior to attachment to the USB host. It then no longer needs to be called.

In LUNS that are marked as removable, the media information is dynamic. The function should still be called at the beginning of execution to indicate the initial state of the media, and then it should also be called every time the media changes.

See Sec. 8.3.4 for more about informing the API of media changes.

Parameters

Table 49. Parameters for *USBMSC_updateMediaInfo()*

BYTE *lun*	The logical unit (LUN) on which the operation is taking place. Zero-based. (This version of the API only supports a single LUN.)
USBMSC_mediaInfoStr* *info*	A structure that communicates the most recent information about the medium.
Returns	*kUSB_succeed*

Table 50. USBMSC_mediaInfoStr

Type	Name	Description
BYTE	mediaPresent	Indicates that the medium is present (non-zero) or not present (zero).
BYTE	mediaChanged	If non-zero, indicates that the medium is not the same one that was present the last time this function was called. (This value is only valid if mediaPresent is non-zero.)
BYTE	writeProtected	Indicates that the medium is write-protected (non-zero) or not write-protected (zero). (This value is only valid if mediaPresent is non-zero.)
BYTE[4]	lastBlockLba	LBA of the last block in the media. (This value is only valid if mediaPresent is non-zero.)
BYTE[4]	bytesPerBlock	Bytes per block. (This value is only valid if mediaPresent is non-zero.)

11 Event-Handling

The USB interrupt service routine (ISR) is contained within the API, and is not intended to be modified. Rather, the API provides *events* the application needs to handle. Most of the defined events are called from within the ISR, in response to various interrupt types.

As with ordinary ISR handlers, the event handlers provide a way to keep the CPU awake after the interrupt service routine has completed. This is a common technique in MSP430 programming.

The handler functions are located in *USB_eventhandling.c.* This file is considered user space, and the software designer has complete freedom to modify these handlers.

11.1 The Relationship between Interrupts and Events

The event handler functions can be thought of as ISR handlers. The software designer can write code into these functions without editing the ISR directly. Using the function *USB_setEnabledEvents()* function can be thought of similarly to setting/clearing interrupt enable bits.

As part of the USB ISR, general interrupts are disabled during the event handler, and therefore any precautions about good ISR coding apply to event handlers as well. For example, it's recommended the handler be kept as short as possible, so that the ISR can finish quickly; this avoid delays in handling other interrupts. (If more functionality is required, see Sec. 11.2 for another means of handling). The event handler should not enable general interrupts, since it opens the door to nested interrupts, which could cause a stack overflow.

11.2 Waking from Event Handlers

MSP430 coding usually makes frequent use of its low-power modes. The program spends most of its time in a low-power mode, experiences an interrupt, and then goes back to sleep. Sometimes the interrupt handling requires that the CPU stay awake after the ISR completes; this can be accomplished with an intrinsic function such as:

```
__bic_SR_register_on_exit(LPM3_bits);
```

Upon exiting the ISR in this case, the CPU will not return to its low-power mode, but rather will stay awake and resume executing in main() from the point the low-power mode had been entered. (This technique is well-documented in other MSP430 material, including the application note *MSP430 Software Coding Techniques (slaa294)*.)

Since events are generally extensions of the USB ISR, they often have a similar need to keep the CPU active. Instead of issuing this intrinsic function, the event handlers return a TRUE or FALSE value to indicate whether the CPU should remain active. Returning a value of FALSE causes the CPU to go back to sleep after the ISR, while a TRUE value causes the CPU to stay awake.

11.3 Calling API Functions from Event Handlers

Generally speaking, it isn't recommended to send data from within an event handler. The main problem is that event handlers occur during an ISR context, and data can't be sent during an ISR. If data is written to be sent from within the event handler, it needs to call *USBCDC_intfStatus()* or *USBHID_intfStatus(),* and it may find that the interface is busy. If it is in fact busy, it won't become available until the event handler exits. This means a proper sending construct needs to assume it may need to send *after* the ISR completes – in other words, in main().

Since a proper application needs to put sending code in main() anyway, it may as well be designed this way from the beginning rather than having some in the event handler and some in main(). Therefore, if data transmission is to be triggered from an event, a flag can be set from within the event handler, and the handler can return TRUE, as described in Sec. 11.2.

All other API calls can be made from within event handlers.

11.4 Using *USB_setEnabledEvents()*

The event handlers only run in response to an event if their respective event is *enabled.* Calling the *USB_setEnabledEvents()* function is analogous to setting/clearing interrupt enable (IE) bits. By default, they are all disabled. Most applications need only call this function once, at the beginning of the program, simply to enable all events that will be needed.

The only reason to keep an event disabled is to avoid wasting execution cycles by calling "empty" event handlers. In some cases these wasted cycles could affect bandwidth.

11.5 Event Handler Functions

A summary of the event handler functions is shown in the table below.

Table 51. Event Handler Summary

Event Handler functions	Interrupt source	Event Description
`BYTE USB_handleClockEvent()`	USBVECINT_PLL_RANGE	USB PLL failed (out of range)
`BYTE USB_handleVbusOnEvent()`	USBVECINT_PWR_VBUSOn	Indicates that a valid voltage is now available on the VBUS pin (i.e., the bus is now attached)
`BYTE USB_handleVbusOffEvent()`	USBVECINT_PWR_VBUSOff	Indicates that a valid voltage is no longer available on the VBUS pin (i.e., the bus has been removed)
`BYTE USB_handleResetEvent()`	USBVECINT_RSTR	Indicates that the USB host has initiated a port reset. This has a similar effect to calling USB_reset(), including that any enumeration will be lost.
`BYTE USB_handleSuspendEvent()`	USBVECINT_SUSR	Indicates that the USB host has put this device into suspend mode.
`BYTE USB_handleResumeEvent()`	USBVECINT_RESR	Indicates that the USB host has resumed this device from suspend mode.
`BYTE USB_handleEnumCompleteEvent()`	Interrupt for control endpoint 0	Indicates that the USB device has just become enumerated.
`BYTE USBCDC_handleDataReceived()`	Interrupt for the output endpoint associated with this data interface	Indicates that data has been received for CDC interface *intfNum*, for which no data receive operation is underway
`BYTE USBCDC_handleSendCompleted()`	Interrupt for the input endpoint associated with this data interface	Indicates that a send operation on CDC interface *intfNum* has just been completed
`BYTE USBCDC_handleReceiveCompleted()`	Interrupt for the output endpoint associated with this data interface	Indicates that a receive operation on CDC interface *intfNum* has just been completed
`BYTE USBHID_handleDataReceived()`	Interrupt for the output endpoint associated with this data interface	Indicates that data has been received for HID interface *intfNum*, for which no data receive operation is underway
`BYTE USBHID_handleSendCompleted()`	Interrupt for the input endpoint associated with this data interface	Indicates that a send operation on HID interface *intfNum* has just been completed
`BYTE USBHID_handleReceiveCompleted()`	Interrupt for the output endpoint associated with this data interface	Indicates that a receive operation on HID interface *intfNum* has just been completed
`BYTE USBMCS_handleBufferEvent()`	Interrupt for the input/output endpoint associated with this data interface	Indicates that the API needs for the application to process a buffer.

11.5.1 BYTE USB_handleClockEvent(void)

Description

This event signals that the output of the USB PLL has failed. This event may have occurred because XT2, the source of the PLL's reference clock, has failed or is unreliable.

If this event occurs, the USB connection will likely be lost. It is best to handle it by calling *USB_disconnect()* and attempting a re-connection.

11.5.2 BYTE USB_handleVbusOnEvent(void)

Description

This event indicates that a valid voltage has been applied to the VBUS pin; that is, the voltage on VBUS has transitioned from low to high.

This usually means the device has been attached to an active USB host. It is recommended to attempt a USB connection from this handler, as described in Sec. 6.3.

11.5.3 BYTE USB_handleVbusOffEvent(void)

Description

This event indicates that a valid voltage has just been removed from the VBUS pin. That is, the voltage on VBUS has transitioned from high to low.

This generally means the device has been removed from an active USB host. It might also mean the device is still physically attached to the host, but the host went into a standby mode; or it was attached to a powered hub but the host upstream from that hub became inactive. The API automatically responds to a VBUS-off event by powering down the USB module and PLL, which is the equivalent of calling *USB_disable()*. It then calls this handling function, if enabled.

11.5.4 BYTE USB_handleResetEvent(void)

Description

This event indicates that the USB host has issued a reset of this USB device. The API handles this automatically, and no action is required by the application to maintain USB operation. After handling the reset, the API calls this handling function, if enabled. In most cases there is no significant reason for the application to respond to bus resets.

11.5.5 BYTE USB_handleSuspendEvent(void)

Description

This event indicates that the USB host has suspended this USB device. It's important that a bus-powered, suspended USB device limit its consumption of power from VBUS during this time. The API automatically shuts down USB-related circuitry inside the MSP430's USB module. However, the application may need to shut down other circuitry drawing from VBUS. This handler is a good place to do this.

See Sec.12.1.2 for a complete discussion about handling suspend.

11.5.6 BYTE USB_handleResumeEvent(void)

Description

This event indicates that the USB host has resumed this USB device from suspend mode. If the device is bus-powered, it is no longer restricted in the amount of power it can draw from VBUS. The API automatically re-enables any circuitry in the MSP430's USB module that was disabled during suspend. The application can use this handler to re-enable other circuitry as well.

11.5.7 BYTE USB_handleEnumCompleteEvent()

Description

This event indicates that the device has just become enumerated. This corresponds with a state change to *ST_ENUM_ACTIVE.*

11.5.8 BYTE USBCDC_handleDataReceived(BYTE intfNum)

Description

This event indicates that data has been received for CDC interface *intfNum* with no receive operation underway. Effectively, the API doesn't know what to do with this data and is asking for instructions. The application can respond by either initiating a receive operation or rejecting the data. Until one of these is performed, USB data reception cannot continue; any packets received from the USB host will be NAK'ed.

Therefore, this event should be handled quickly. A receive operation cannot be started directly out of this event, since *USBCDC_receiveData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the receive operation. After this function exits, a call to *USBCDC_intfStatus()* for this CDC interface will return *kUSBDataWaiting*.

If the application is written so that a receive operation is always begun prior to data arriving from the host, this event will never occur. The software designer generally has a choice of whether to use this event as part of code flow (initiating receive operations after data is received), or to always keep a receive operation open in case data arrives. (See Sec. 12 for more discussion.)

11.5.9 BYTE USBCDC_handleSendCompleted(BYTE intfNum)

Description

This event indicates that a send operation on CDC interface *intfNum* has just been completed.

In applications sending a series of data blocks, the designer may wish to use this event to trigger another send operation. This cannot be done directly out of this event, since *USBCDC_sendData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the operation.

11.5.10 BYTE USBCDC_handleReceiveCompleted(BYTE intfNum)

Description

This event indicates that a receive operation on CDC interface *intfNum* has just been completed, and the data is therefore available in the user buffer assigned when the call was made to *USBCDC_receiveData().* If this event occurs, it means that the entire buffer is full, according to the size value that was requested.

The designer may wish to use this event to trigger another receive operation. This cannot be done directly out of this event, since *USBCDC_receiveData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the operation.

11.5.11 BYTE USBHID_handleDataReceived(BYTE intfNum)

Description

This event applies to HID-Datapipe only, as opposed to HID-Traditional. It indicates that data has been received for HID interface *intfNum* with no receive operation underway. Effectively, the API doesn't know what to do with this data and is asking for instructions. The application can respond by either initiating a receive operation or rejecting the data. Until one of these is performed, USB data reception cannot continue; any packets received from the USB host will be NAK'ed.

Therefore, this event should be handled quickly. A receive operation cannot be started directly out of this event, since *USBHID_receiveData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the receive operation. After this function exits, a call to *USBHID_intfStatus()* for this HID interface will return *kUSBDataWaiting*.

If the application is written so that a receive operation is always begun prior to data arriving from the host, this event will never occur. The software designer generally has a choice of whether to use this event as part of code flow (initiating receive operations after data is received), or to always keep a receive operation open in case data arrives. (See Sec. 12 for more discussion.)

11.5.12 BYTE USBHID_handleSendCompleted(BYTE intfNum)

Description

This event applies to HID-Datapipe only, as opposed to HID-Traditional. It indicates that a send operation on data interface *intfNum* has just been completed.

In applications sending a series of large blocks of data, the designer may wish to use this event to trigger another send operation. This cannot be done directly out of this event, since *USBHID_sendData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the operation.

11.5.13 BYTE USBHID_handleReceiveCompleted(BYTE intfNum)

Description

This event applies to HID-Datapipe only, as opposed to HID-Traditional. It indicates that a receive operation on HID interface *intfNum* has just been completed, and the data is therefore available in the user buffer assigned when the call was made to *USBHID_receiveData().* If this event occurs, it means that the entire buffer is full, according to the size value that was requested.

The designer may wish to use this event to trigger another receive operation. This cannot be done directly out of this event, since *USBHID_receiveData()* cannot be called from the event handlers. However, the handler can set a flag for main() to begin the operation.

11.5.14 BYTE USBMSC_handleBufferEvent(VOID)

Description

This event occurs when the API requests a buffer. Immediately prior to this, the API sets the *operation* field of the *USBMSC_RWBuf_Info* structure corresponding with the request, and also clears the low-power-mode bits of the MCU's status register to ensure the CPU remains awake to process the buffer after the event occurs. (Note that this means the return value of this event has no effect; the CPU will remain awake even if this function returns FALSE.)

12 Practical Matters: Writing USB Programs with the API

The discussion in this section applies to any MSP430 CDC or HID device.

12.1 Power Management

The flexibility of the MSP430's USB power system provides options to the system designer. As a result, arrangement of power is primarily the application's responsibility. Software does have a few USB-related responsibilities, however.

12.1.1 API's Control of Power

The API exerts minimal control upon power configuration, beyond initial configuration of the integrated LDO regulator.

USB provides 5V of power via the cable, called VBUS. The MSP430's USB module has an integrated LDO regulator that reduces this to 3.3V. The output of this regulator is called VUSB, and this rail sources the USB module directly. The hardware engineer can also direct this power into DVCC, which sources the rest of the MSP430.

The API configures the USB module such that the LDO automatically activates when 5V bus power is applied to VBUS. Thus it remains enabled the entire time VBUS power is available, and is automatically disabled when VBUS is removed.

12.1.2 Managing VBUS Power During USB Suspend

When the host suspends a USB device, the device (meaning the entire physical USB device, not just the MCU) must reduce the current it's drawing from the host via VBUS. It must do this within 7ms of the suspend event, which the application sees as a *handleSuspendEvent().* It is from this event handler that the application should insert any necessary code to shut down circuitry.

According to the USB specification, the amount most devices are allowed to draw is 500uA. However, the USB Implementers Forum reliably gives waivers to any device that consumes in the range of 500uA to 2.5mA.

As discussed in Sec. 6.5, the USB module's power is handled automatically. If DVCC is drawing its power from VUSB (the output of the internal 3.3V regulator, usually derived from VBUS), the application is responsible for shutting down peripherals and/or CPU in order to meet the requirement. The device datasheet provides the parameters necessary to determine the budget. Software may also need to disable components elsewhere in the system; for example, if a battery is being charged from USB, this will need to be disabled.

If a crystal is used on XT2, the oscillator consumes a couple hundred uA's. The Descriptor Tool can configure the API to attempt to power down XT2 during suspend. However, if any peripherals derive their clocking from XT2, the oscillator will remain on, due to the peripheral issuing a "clock request". (See the F5xx Family User's Guide for more information.)

Powering down circuitry may very well mean an overall change in state for the device. If so, the code may benefit from the state-dependent implementation as described in Sec. 12.5. It is important to consider that a suspend can happen at any point in the code flow. Therefore, it's usually important to build the code so that it can exit from any event when the USB state has changed.

When the host "resumes" the device from suspend, it is probably desirable for software to re-activate the components that were disabled. This can be performed within *handleResumeEvent().*

In battery-powered systems, the engineer is encouraged to draw most of the device's power from VBUS while it's available. Being attached to a USB host may keep the device active for longer periods of time than would occur for a non-USB mobile device, so if VBUS power is not used, the battery will experience extra power draw.

12.1.3 Use of Low-Power Modes

The software engineer is encouraged to use the MSP430's low power modes in the application, because maximizing the time spent in these modes can increase power efficiency. They can be used similarly to any MSP430 application. The only restriction is that while the USB connection is enumerated and active, the power mode should not be "less" than LPM0 (in other words, do not use LPM3/4/5). During USB suspend, it can be placed into LPM3 (but not LPM4/5).

LPM entry often needs to be conditional upon USB events. Since USB events are typically interrupt-driven, it is recommended to disable interrupts prior to evaluating these conditions, and re-enabling flags atomically with the LPM entry:

```
__disable_interrupt();
if(!dataReceivedFlag)
  __bis_SR_register(LPM0_bits + GIE); //Simultaneous LPM0 entry & re-enabling ints
__enable_interrupt();                 //Enable ints if LPM0 was not entered
```

In this example, *dataReceivedFlag* is set in *USBxxx_handleDataReceivedEvent().* If interrupts had not been disabled, then it would have been possible for the flag to have changed state after the evaluation but before the LPM entry. If this were to happen, the LPM entry would be in error, and the device may possibly never wake from LPM0. Disabling interrupts before the evaluation ensures the event won't occur.

12.2 Clock System Management

As with power, arrangement of clocks in an MSP430 USB application is the application's responsibility. However, the API takes some actions that affect clock management, of which the software engineer should be aware.

12.2.1 Summary of USB Clock System

MSP430 devices supporting USB include a PLL that generates a 48MHz USB clock from a reference clock sourced by the XT2 oscillator. Unlike other MSP430 peripherals, the PLL is sourced directly from XT2, rather than via a system clock such as SMCLK.

During USB suspend, the PLL automatically switches its reference clock to the VLO (very low-frequency, low-power oscillator). This low-power oscillator is sufficient for the module's functions during suspend, and XT2 is not needed at this time.

12.2.2 Sourcing the USB PLL Reference Clock

The simplest method to source the PLL reference is placing a crystal on XT2. An alternative that might reduce cost is to inject a clock from outside the MSP430 into XT2, with XT2 placed in "bypass mode".

The PLL reference must adhere to certain frequency, precision, and jitter requirements. (See the device datasheet.) Crystals on XT2 easily meet these requirements.

If bypass mode is desired instead, the designer must consider these requirements. The tolerance requirement is driven by the USB specification: ±2500ppm. The MSP430's DCO/FLL output does not meet the jitter requirements, and thus it cannot be used for USB. The low-frequency oscillators on the device (LFXT1, REFO, and VLO) do not run at a high enough frequency to source XT2 for USB.

Therefore, the PLL reference must come from either:

- a crystal on XT2, or
- a clock source from elsewhere on the board – injected into XT2 in bypass mode – that originates from a crystal or perhaps some ceramic resonators capable of the needed precision

12.2.3 Configuring the API for the Clock Conditions

The API takes these actions with respect to clock configuration:

- It automatically starts XT2 when it needs it
- It configures the PLL's reference frequency

USB cannot function if XT2 isn't active; so the API activates it in *USB_enable().* If XT2 isn't being used for other functions, the application may choose to disable it in *USB_handleVbusOffEvent(),* saving power.

The Descriptor Tool can configure the stack to disable XT2 during suspend. If it does, it will also automatically re-start it when resuming. Note that it can only "attempt" this deactivation, because if any MSP430 peripherals (or CPU) are using XT2, they will automatically keep XT2 active by issuing a "clock request". (See the *F5xx Family User's Guide* for more information.)

The frequency the API configures for the PLL reference can be selected with the Descriptor Tool. Its value depends on the crystal chosen for XT2 (or the external clock source used for XT2 in bypass mode).

The API has one other aspect of clock interaction. In a few places, the API employs CPU-driven delay loops. CPU speed (called MCLK) is chosen by the application, and so in order to prevent the speed from changing the delay period of the loops, they take MCLK into account. They do this via a compile-time constant generated by the Descriptor Tool. Hence, it's important that the

system designer enter this value. If MCLK isn't always the same value during USB operation, choose the fastest value that is ever in effect; this will result in a delay that is longer than necessary, in some cases, but it is safe compared with choosing a delay value that is too small.

The API stack project includes a set of functions that assist in setting up the F5xx Unified Clock System (UCS) module. These are contained in *hal_UCS.c.*

12.2.4 XT2 Startup Times

It's important that the load capacitors for XT2 be chosen properly, according to the crystal's rating. An improperly-tuned oscillator circuit will result in either oscillator failure, or extended startup times. Startup times are important if XT2 is disabled during suspend, because once the host resumes the USB device, it has 10ms to begin communicating with the host. A properly-tuned XT2 oscillator circuit stabilizes well within this range, but an improperly-tuned one can take significantly longer.

12.2.5 Using XT2 for Non-USB Functions

It's possible for XT2 to be used for non-USB functions; for example, it can drive the CPU, via the system clock MCLK; and it can drive peripherals, via the system clock SMCLK. However, this is only recommended if there's a specific need for crystal accuracy in these functions. In particular, it is best to drive MCLK from the DCO. There are several reasons for this.

One is that some earlier USB-equipped MSP430 devices have an errata that causes "bus errors" to occur if the DCO isn't used to drive any system clocks. (See next section for information on bus errors.) On these devices, USB won't function unless the DCO is active.

Another is that if XT2 is used for other functions, then it may need to run during USB suspend. The DCO consumes much less current than XT2 does. If the device's DVCC power rail is sourced from VBUS, XT2's current draw will count against the VBUS current budget.

If any MSP430 component is configured to use XT2 (via MCLK or SMCLK), then XT2 will remain enabled whenever it's needed by that component. The API will not be able to disable it, due to "clock requests" that keep it active.

12.2.6 "Bus Errors"

As mentioned earlier, the USB module is clocked by an internal PLL. During USB suspend, the PLL is disabled, and the USB module is clocked by the VLO oscillator. The VLO operates at a much slower frequency than the PLL. If the CPU attempts to access the USB RAM at this time, a non-maskable interrupt (NMI) will occur, called a "bus error". (USB RAM must be accessed anytime there is an active USB connection; see the F5xx Family User's Guide for details.)

USB RAM is "multi-port" – that is, it's mutually accessible by the USB module and the CPU, and thus affected by both clock domains. The gap in speed between MCLK and the VLO will cause timeouts to occur on the MSP430's internal data bus if an access is attempted. This generates a bus-error NMI.

Bus error NMIs must be handled by resetting the USB module, with *USB_disable().*

```
case SYSUNIV_BUSIFG:
  SYSBERRIV = 0;
  USB_disable();
```

To re-attempt the USB connection, subsequent calls to *USB_enable()* and *USB_connect()* are required.

A common cause of a bus error is entering LPM3 while the USB state is ST_ENUM_ACTIVE. If this isn't the cause, then usually the problem somehow involves the PLL being inactive at a time the software thinks it's active (and thus attempting to access USB RAM).

12.2.7 Use of DMA

USB transfers will generally occur more quickly and efficiently if DMA is enabled. This can be configured using the Descriptor Tool.

12.2.8 Using an RTOS

The MSP430 USB API stacks were designed to not require an RTOS. As such, the discussion in this section is written from the perspective that a main loop is employed. However, the API stacks are intended to be straightforward to port to an RTOS, and most of the concepts in this section still apply to an RTOS-based system.

12.3 System Interrupts

USB operation involves a large number of interrupts. In some cases these interrupts have timeout periods enforced by the host, and if the USB device takes too long to respond, USB operations may fail. USB interrupts fall below several other kinds of interrupts on the MSP430's interrupt priority list – specifically the ones shown in the table below.

Table 52. Interrupts with Higher Priority than USB

Interrupt Source	Priority
System Resets	63 (highest)
System/User NMIs	61-62
Comparator_B	60
TB0	58-59
Watchdog Timer (interval timer)	57
USCI_A0/B0 (SPI/I2C/UART)	55-56
ADC12_A (12-bit A/D)	54
Timer_A	52-53
USB	51
Other interrupts	0-50

If these other interrupts occur with unusually high frequency in the application, or with high duration, they might prevent the USB interrupts from being serviced.

12.4 USB Design Considerations

12.4.1 Robustness: Handling Surprise Removal or Suspend

It's important for the software designer to consider that a USB connection can be lost at any time, due to being suspended by the host, or by the user disconnecting it (a "surprise removal"). A device should recover from these events gracefully. The API functions and the example constructs provide help in this regard, having been designed to return the necessary information.

12.4.2 The Impact of USB State on Functionality

As discussed in Sec. 6, a USB device can be in one of several different bus states, returned by *USB_connectionState()*. A device spends most of its time in ST_DISCONNECTED, ST_ENUM_ACTIVE, and ST_ENUM_SUSPENDED. The state can change at any time.

For many USB devices, these states have a deep impact on functionality. For example, a digital still camera's main function when not connected to a host is to take pictures. But once connected to the host, its purpose is now to upload pictures. It may even prevent pictures from being taken. Thus the state of the bus has completely altered the software flow. This is in contrast to most equipment that uses an RS232 serial port; such devices typically do not change "identity" according to the connection state, the way many USB devices do.

Not all USB devices behave this way. Devices that are purely bus-powered (like a mouse or keyboard) do nothing when not attached to a host. And some devices keep essentially the same functionality whether attached to USB or not – the only difference is whether these devices attempt to upload data.

USB state may also affect the power configuration. A battery-powered device may rely on VBUS for power while USB is attached, which means it needs to significantly reduce functionality when the host suspends it.

Several example devices are shown in the table below, describing how they behave in the three primary states.

Table 53. State-Dependence of Some Example Devices

	ST_DISCONNECTED	ST_ENUM_ACTIVE	ST_ENUM_SUSPENDED
Digital camera	Functions as a camera	Playback and upload; pictures can't be taken	Functions as a camera
Cell phone	Functions as a phone	Enumerates as a mass storage device; phone calls can't be made	Functions as a phone; stops charging the battery
Mouse	Unpowered	Functions as a mouse	Does nothing, except may be capable of remote wakeup
Printer	Allows configuration from its control panel	Allows configuration, and also responds to print requests	Allows configuration from its control panel

Therefore, one of the first decisions the designer should consider is how (or if) the USB state will impact functionality.

12.5 State-Dependent Functionality: Main Loop Framework

For devices that change functionality significantly according to USB state, one option for a main loop construct is shown below.

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;                   // Configure watchdog
    Init_System();
    USB_init();
    USB_setEnabledEvents(kUSB_allUsbEvents);

    if (USB_connectionInfo() & kUSB_vbusPresent)
      USB_handleVbusOnEvent();

    while(1)
    {
        switch(USB_connectionState())
        {
          case ST_USB_DISCONNECTED:
             __bis_SR_register(LPM3_bits + GIE);  // Enter LPM3
             function_as_USB_disconnected();
             break;
          case ST_USB_CONNECTED_NO_ENUM:
             break;
          case ST_ENUM_ACTIVE:
             __bis_SR_register(LPM0_bits + GIE);  // Enter LPM0
             function_as_USB_active();
             break;
          case ST_ENUM_SUSPENDED:
             __bis_SR_register(LPM3_bits + GIE);  // Enter LPM3
             break;
          case ST_ENUM_IN_PROGRESS:
             break;
           case ST_NOENUM_SUSPENDED:
             break;
          case ST_ERROR:
             break;
          default:;
        }
     }
}

BYTE USB_handleVbusOnEvent()  {
    if (USB_enable() == kUSB_succeed)  {
        USB_reset();
        USB_connect();
    }
    return TRUE;                // Wake the main loop to give it a chance
}                               // to respond to the change in state

BYTE USB_handleVbusOffEvent()  {
    return TRUE;                // Wake the main loop to give it a chance
}                               // to respond to the change in state

BYTE USB_handleSuspendEvent()  {
    return TRUE;                // Wake the main loop to give it a chance
}                               // to respond to the change in state

BYTE USB_handleResumeEvent()   {
    return TRUE;                // Wake the main loop to give it a chance
}                               // to respond to the change in state
```

Execution within this main loop “forks” depending on the state of the USB, creating several alternate main loops. As such, USB state becomes a central part of managing software flow.

When this device isn’t attached to USB, the main loop functionality is defined by `function_as_bus_disconnected()`. When the device is attached, *handleVbusOnEvent()* initiates a connection. (There is no rule that says a USB connection must be made upon attachment; so if this isn’t desired, this code could be removed.)

While waiting for enumeration to complete, the state is ST_ENUM_IN_PROGRESS. When enumeration completes, the main loop waits for an interrupt to begin executing `function_as_bus_active()`.

handleVbusOnEvent() is also called at the top of main(). Normally the application doesn’t call the event handler functions; the API does this. However, this is an exception. Since *handleVbusOnEvent()*only gets called when VBUS transitions, it misses the case where VBUS is already on when execution begins. Thus it must be done “manually”.

Notice that these four events are enabled and set to return TRUE:

- *USB_handleVbusOnEvent()*
- *USB_handleVbusOffEvent()*
- *USB_handleSuspendEvent()*
- *USB_handleResumeEvent()*

This is because it’s important the main loop be “refreshed” to reflect the new state. Each of these events correlates with a state change. If these handlers execute and return with a TRUE return value, they will cause the main loop to execute once. If the loop is asleep at an LPM instruction inside one of the `case` statements, this allows it to leave the main loop and make another call to *USB_connectionState().* It can then handle whatever functionality is required in the new state.

Note that the case for ST_ENUM_IN_PROGRESS does not enter an LPM mode. This keeps the main loop active for the brief time until the state changes to ST_ENUM_ACTIVE. The purpose of this is the same as returning TRUE in the event handlers – it gives the main loop a chance to respond to the forthcoming change in state.

If it’s desired for a device to behave the same way in two separate states – for example, whether suspended or disconnected – their ”case” lines could be combined.

Complete code files for *main.c* and *USB_eventHandling.c* corresponding with this approach are included in the API source distribution, under “Recommended Main Loop Framework”. It is fully commented to explain the design rationales.

12.6 State-Independent Functionality

For devices that don’t significantly change their functionality according to USB state, something similar to below could be employed.

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;            // Configure watchdog
    Init_System();
    USB_init();
    USB_setEnabledEvents(kUSB_allUsbEvents);

    //Connect to USB if cable already was plugged in
    if (USB_connectionState() == ST_USB_CONNECTED_NO_ENUM)
        USB_handleVbusOnEvent();

  while(1)  {
    if(USB_connectionState() == ST_ENUM_ACTIVE)   // Enter a low-power mode
      __bis_SR_register(LPM0_bits + GIE);        // LPM3 if possible, but must
    else __bis_SR_register(LPM3_bits + GIE);      // stay in LPM0 if USB is active

    process_data();                               // Do what we woke up to do

    if(USB_connectionState() == ST_ENUM_ACTIVE)   // If USB happens to be active,
    {                                             // also send the data
      prepare_buffer(buffer,size);
      USBCDC_sendData(buffer,size,0);
    }
  }  // while(1)
} //main()


BYTE USB_handleVbusOnEvent()
{
    if (USB_enable() == kUSB_succeed)  {
        USB_reset();
        USB_connect();
    }
    switchPowerToVBUS();
    return FALSE;
}

BYTE USB_handleVbusOffEvent()
{
    switchPowerToBattery();
    return TRUE;
}

BYTE USB_handleSuspendEvent()
{
    switchPowerToBattery();
    return FALSE;
}

BYTE USB_handleResumeEvent()
{
    switchPowerToVBUS();
    return FALSE;
}

#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void) {
  __bic_SR_register_on_exit(LPM3_bits);            // Exit LPM
}
```

This example wakes periodically and processes data. If USB happens to be available, it also broadcasts the data to the host. The functionality is largely the same whether USB is attached or not. Implementing this with the earlier `switch()` loop may have resulted in considerable duplication of code.

In this model, USB state must be checked on a case-by-case basis as USB connectivity is needed. Notice that it must also be checked before entering LPM3; doing so during USB active would result in a "bus error" non-maskable interrupt (see Sec.12.2.6). It must be checked again later before sending data.

This model relies on events more than the state-dependent model did. This example portrays a battery-powered device. Staying active on USB requires considerable power, so it must switch its power source to VBUS if it's available. However, if it's not available, it switches its power source back to the battery so that it can continue its normal operation. It uses events to accomplish this switching.

Notice that this time, the events return FALSE. The main loop in this case doesn't require any "refreshing", as the state-dependent loop did. However, one event returns TRUE: *handleVbusOffEvent().* The reason is that if this event occurs while the device is in LPM0, the reason for avoiding LPM3 is no longer valid, and depending on the timer period, this might end up wasting an appreciable amount of power. Waking from LPM0 allows the loop to run once, and this time enter LPM3. (This concept is very similar to why event handlers returned TRUE in the state-dependent `switch()` loop.)

This particular example doesn't check for pre-existing send operations before calling *USBCDC_sendData()*, as arguably it should. However, if one already exists, the function simply returns an error and execution continues. The only consequence is that this particular data transmission was missed.

12.7 Tips on Sending Data over CDC or HID-Datapipe

The API provides the functions *USBCDC_sendData()* and *USBHID_sendData().* When using them, two things need to be considered:

- *Background Processing* (or "asynchronous" operation). Send operations can continue to execute after *sendData()* returns. Software must anticipate that a previous operation may still be in progress, before making another call to *sendData().*
- *Bus disconnect/suspend*. The end user may disconnect the bus at any time, or the host may suspend the device at any time. Software must anticipate that the bus may suddenly become unavailable.

TI provides example construct functions that take these into account. The developer is free to edit them, but they function well in most applications as-is.

Equivalent functions are provided for both CDC and HID-Datapipe, reflecting the similarity between these two interfaces. The function prefixes are *cdc* or *hid,* accordingly. Since the text in this section applies equally to both types, they're frequently shown in this section with the prefix *xxx* -- for example, the text "*cdcSendDataInBackground()* and *hidSendDataInBackground()"* is shortened to "*xxxSendDataInBackground()"*.

The functions are included with the application examples, in the file *usbConstructs.c/h.*

12.7.1 Background Processing

As discussed in Sec. 7.2.2, background send/receive operations increase efficiency, and they ensure fluid execution no matter what happens on the host. The tradeoff is that if more than one send attempt will be performed, software must be mindful that a previous operation can still be in progress.

If desired, background processing can be limited. This is done by iteratively polling the status of the interface (*USBxxx_intfStatus()*) -- either after the call to *USBxxx_sendData()*, or before.

If doing it after, this is called *post-call polling.* It prevents execution from proceeding past the point of initiating the send until all the data has been transferred. This effectively eliminates background processing altogether, which makes coding simpler in some respects. However, it has two disadvantages: it wastes CPU cycles during polling, and it also has a tendency to "lock" the MCU's code execution to the host ("synchronous" operation). If the host and bus are "fast", this synchronicity isn't a problem. But host/bus performance are outside the device's control, and can't be guaranteed.

An example of synchronicity being problematic is if the host is writing large amounts of data to a USB hard drive located on the same bus, making the host and/or bus "slow". In this situation, synchronous operation (post-call polling) might impact MCU code execution.

Another approach is *pre-call sending.* Data is sent using *USBxxx_sendData(),* and then execution immediately resumes while the API sends the data in the background. The next time the application wants to send data, however, it iteratively polls *USBxxx_intfStatus() before* the next send attempt. The advantage over post-call polling is that much of the data -- if not most, or all -- has probably already been sent in the background by the time this occurs. As a result,

it's usuallly much less wasteful. At the same time, it can easily "contain" background processing, while still providing the simple coding of issuing a single function call.

There are two disadvantages to pre-call polling. First, the application isn't allowed to edit the user buffer associated with the previous send operation, unless it's sure that operation has been completed in the background. (However, this can easily be handled by using an X/Y buffer scheme -- see Sec. 12.7.4.) The second is that it can still allow wasted cycles if sending didn't finish in the background before the next send. But in most situations this isn't the case; the polling is mostly eliminated.

A third approach is true "asynchronous" sending. It involves no polling of *USBxxx_intfStatus(),* but rather makes use of the API's events to allow the CPU to sleep or perform other functions until the *USBxxx_handleSendCompleted()* event occurs, only then triggering the next send attempt. Since this approach involves no polling, it's somewhat more efficient than pre-call polling, at the expense of being a little more difficult to code.

The three approaches are summarized below.

Table 54. Three Approaches to Sending Data

	Description	Example Construct Function Provided in API
Post-call polling (no background processing)	Poll the interface status **after** *USBxxx_sendData(),* until the operation is complete.	*cdcSendDataWaitTilDone()* *hidSendDataWaitTilDone()*
Pre-call polling ("Contained" background processing)	Poll the interface status **before** *USBxxx_sendData()*, until the interface is available.	*cdcSendDataInBackground()* *hidSendDataInBackground()*
True "asynchronous" operation	No interface polling. Instead, call *USBxxx_sendData()* after a *USBxxx_handleSendCompleted()* event for the previous send; and/or perform a single interface check before sending.	*Not applicable*

Table 55. Approaches to Sending Data

	Cycle Efficiency	Fluid Execution (not synchronous with host)	Ease of Use
Post-call polling	Low	Low	High
Pre-call polling	Good	Good	Moderate
True "asynchronous" operation	Excellent	Excellent	More difficult

Here are some guidelines:

- *xxxSendDataWaitTilDone()* can generally be used for small transfers without negative consequence. If fast development is the priority, and the system has cycles to spare, this might be the best approach.

- *xxxSendDataInBackground()* provides a very good balance of efficiency and ease of use. It is also much more asynchronous with the host. Therefore, if performing larger transfers, pre-call polling might be better, because it maximizes bandwidth and minimizes dependency on the host.
- True asynchronous operation might be desirable in RTOS-based implementations that want to completely eliminate the possibility of polling, or if maximum CPU efficiency is required.
- Use caution if mixing pre- and post-call polling. For example, if using *xxxSendDataWaitTilDone()* after a previous call to *xxxSendDataInBackground(),* be sure to poll *USBxxx_intfStatus()* to ensure the previous send operation is complete.

12.7.2 Anticipating a Lost Bus

The USB connection might be disconnected or suspended at any time; and as Sec. 12.4 indicated, the program should respond to these changes in USB state. For example, it's undesirable for the program to "hang up" in a location that no longer fits the current state of the bus because it made an incorrect assumption about bus availability. By checking return values, software can break from the main loop and re-assess the situation.

The construct functions return a value that indicates whether the bus is available or not, so that the application may choose to handle a lost bus. For example, software may need to close down certain operations. If this occurs and execution is in a section that is very USB-specific, it should break from this location.

12.7.3 Constructs: cdcSendDataWaitTilDone() and *hidSendDataWaitTilDone()*

These construct functions do not return until the send operation is completed, eliminating background processing. When it returns a zero value, the send operation has returned successfully. Because of this, the user buffer can be edited immediately after the function returns, without consequence.

It assumes the interface is available (no active send operations) when the function is called. If all sending is performed with this function, then this assumption will always be valid.

If the function returns with a non-zero value, the transfer failed. The operation should be aborted, and since the bus may no longer be available, the application may need to break from its current context.

This call is recommended if all transfers are small, and bandwidth is not a concern. It is robust, anticipating any bus- or host-availability issues.

```
while(1)
{
    switch(USB_connectionState())
    {
       case ST_ENUM_ACTIVE:
            {
               .
               .
               dataBuffer = temporaryBuffer1;
               if(cdcSendDataWaitTilDone(dataBuffer,100,1,0)
               {
                 USBCDC_abortSend(&x,1);
                 break;
               }
               .
               .
               dataBuffer = temporaryBuffer2;
               if(cdcSendDataWaitTilDone(dataBuffer,100,1,0)
               {
                 USBCDC_abortSend(&x,1);
                 break;
               }
               .                          Operations complete when
               .                          sendData_waitTilDone() returns
               .
            }
     }
}
```

Figure 18. Usage of *cdcSendDataWaitTilDone()*

In this example, two send operations are initiated in succession. Notice that *dataBuffer* is edited between the two calls, without needing to be concerned that the buffer is still in use. This is because *xxxSendDataWaitTilDone()* polled until the operation was complete, before returning.

Notice that each call checks for a non-zero return value. Such a value would indicate a serious bus issue – either the bus has become unavailable, or the host for whatever reason isn't cooperating. In each case, the operation is aborted.

The parameters for the construct functions are the three for *USBxxx_sendData()* and also one for a timeout value. This timeout mechanism is an inexact mechanism, but it effectively ensures that executation doesn't stay here forever. The value is very affected by CPU clock speed, since it merely represents the number of retries. Large timeout values are recommended, in order to give the host every chance to respond before concluding that it is inactive.

See Sec. 14 for a complete definition of *xxxSendDataWaitTilDone().*

12.7.4 Constructs: *cdcSendDataInBackground()* and *hidSendDataInBackground()*

This construct function first polls to ensure that the interface is available, and then calls *USBxxx_sendData().* It then immediately returns. When it returns a zero value, the send operation has *begun* successfully, but may not have been completed. Because of this, the user buffer should not be edited until a call to *USBxxx_intfStatus* shows the operation is complete.

If the function returns with a non-zero value, the transfer failed. The operation should be aborted, and since the bus may no longer be available, the application may need to break from its current context.

This call takes full advantage of background operation, increasing efficiency in large-data transfers. It is robust, anticipating any bus- or host-availability issues.

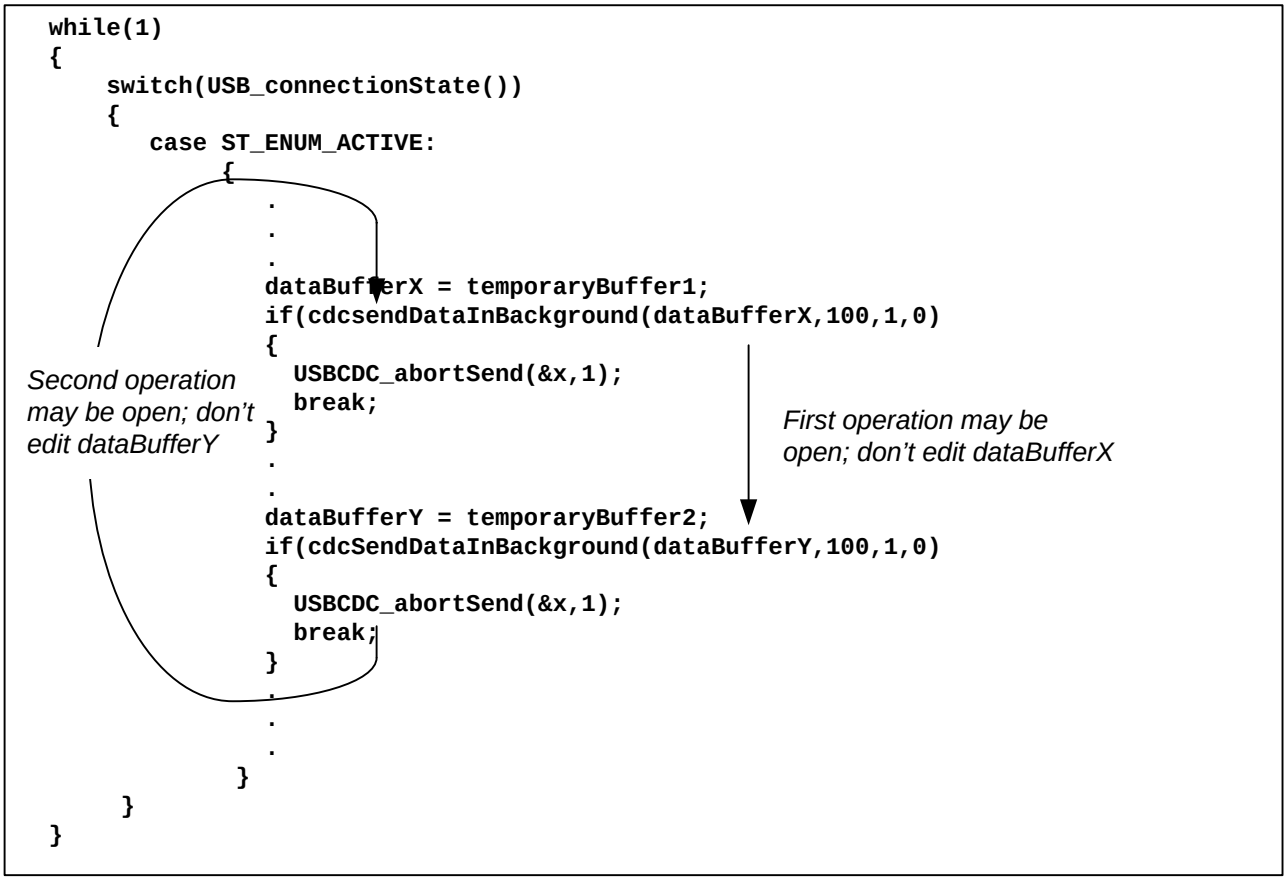


Figure 19. Usage of *cdcSendDataInBackground()*

Two send operations are initiated in succession. Notice that two different buffers are used – *dataBufferX* and *dataBufferY.* Each buffer is not allowed to be edited while its operation may still be underway. Effectively, then, this arrangement uses an switched X/Y buffer scheme.

Notice that each call checks for a non-zero return value. Such a value would indicate a serious bus issue – either the bus has become unavailable, or the host for whatever reason isn't cooperating. In each case, the operation is aborted.

As with *xxxSendDataWaitTilDone(),* the last passed-in parameter is a timeout value. Large values are recommended.

See Sec. 14 for a complete definition of *cdcSendDataInBackground()* and *hidSendDataInBackground().*

12.8 Tips on Receiving Data over CDC or HID-Datapipe

Receiving USB data requires different considerations than sending. When the application sends data, the operation will be completed as fast as the bus and host allow. When receiving data, the application may not know:

- how much data will be sent, or
- when it will be sent

These two factors create a need for a wider range of constructs than for send operations. Many variations can be created, but the ones described below cover most usage scenarios.

Table 56. Receive Construct Options

	Size is Known?	Expected Size	Description
cdcReceiveDataInBuffer() **hidReceiveDataInBuffer()**	No	Small	Receive operations aren't opened until data is already waiting in the USB endpoint buffers. Therefore, the operations are immediately completed. The application must always be available to respond to *handleDataReceived().*
Continuously-Open Receive	No	Large	The application maintains an open, larger-than-needed receive operation at all times. When it wants to use the data, it simply looks inside the user buffer.
Fixed Receive	Yes	n/a	A receive operation is opened for the exact amount of data expected. When this amount is received, a *handleReceiveCompleted()* event is generated, and the application handles it.

As with the sending constructs described in the previous section, matching receive construct functions are provided for both CDC and HID-Datapipe. The function prefixes are *cdc_* or *hid_,* accordingly. Since the text in this section applies equally to both types, they're frequently shown in this section with the prefix *xxx_*.

The functions are included with the application examples, in the file *USB_constructs.c/h.*

12.8.1 cdcReceiveDataInBuffer() and *hidReceiveDataInBuffer()*

The application may choose to call the construct function *xxxReceiveDataInBuffer()* rather than *USBxxx_receiveData().* This function opens a receive operation only for data that has already been received into the USB endpoint buffers, which immediately completes. (Recall that the endpoint buffers contain a single packet of data from the host – see Sec. 7.) This method works best when the expected block of data is small in size, but the exact number of bytes is unknown.

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;                  // Configure watchdog
    Init_System();
    USB_init();
    USB_setEnabledEvents(kUSB_dataReceivedEvent);

    //Connect to USB if cable already was plugged in
    if (USB_connectionState() == USB_CONNECTED_NO_ENUM)
        USB_handleVbusOnEvent();

    while(1)
    {
      switch(USB_connectionState())
      {
        case ST_ENUM_ACTIVE:
          if(!bDataReceived_event)
            __bis_SR_register(LPM0_bits + GIE);              // Enter LPM0

          // CPU was awakened from LPM0
          while(bDataReceived_event)
          {
            bDataReceived_event = FALSE;
            BYTE count = cdcReceiveDataInBuffer(buffer,size,0); // Fetch contents of buffer
            process_the_data(buffer,count);                  // Process them
          }
          break;
          .
          .
          default:;
        }
    }  // while(1)
} //main()


BYTE USBCDC_handleDataReceived(BYTE intfNum)
{
    bDataReceived_event = TRUE;     // Signal that data has been received
    return TRUE;                    // Keep CPU awake after returning from event
}
```

Figure 20. Usage of *cdcReceiveDataInBuffer()*

USBxxx_handleDataReceived() is used to set a flag that data is available in the USB endpoint buffers. The event handler returns TRUE to signal the CPU to remain awake. As a result, execution proceeds from the LPM0 entry in main.

The flag is evaluated by a while() loop. This is because more data might be received during this branch of code, which would need to be handled before entering LPM0 again. It's for this same reason that the flag is immediately cleared before fetching the data. If *xxxReceiveDataInBuffer()* gets called and there are no bytes there, no harm is done, so it's better to call it more times than necessary than to miss data.

The application could have chosen to simply poll *xxxReceiveDataInBuffer,* rather than set a flag. The problem with this is that more data might be received while the last data is being processed. For this same reason, the flag gets evaluated immediately before re-entering LPM0.

This approach requires that the data be fetched from the USB buffer quickly. Otherwise, if the host tries to send more data, the device will begin NAKing the host (refusing to accept more data).

A good application of the *xxxReceiveDataInBuffer()* approach is receiving text entry from a terminal application on the host. The number of incoming bytes isn't known, but it will be small. Using this method, a few characters can be received at a time, and accumulated into a longer string. This is demonstrated in several of the application examples that accompany the API.

See Sec. 14 for a complete definition of *cdcReceiveDataInBuffer()* and *hidReceiveDataInBuffer().*

12.8.2 Continuously-Open Receive

This approach keeps an "open ear" at all times. A single call is made to *USBxxx_receiveData()* immediately after enumeration, of a size larger than the amount of expected data. All received data is automatically moved to the user buffer as it's received. Then, at a time convenient to the application, it goes to the user buffer and processes the data.

Unlike *xxxReceiveDataInBuffer()*, the application isn't pressured to find a place for the data when it comes in, since the API has "standing orders" to deposit the data into the user buffer. The user buffer should be large enough that it won't run out of space. When done, the application can re-open another large operation, and the cycle continues.

Notice that the application isn't obligated to respond quickly to received data, as in reactive receive. This approach is therefore favorable to situations where the application isn't able to respond quickly.

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;
    Init_Ports();
    Init_Clock();
    USB_init();
    USB_setEnabledEvents(kUSB_VbusOnEvent + kUSB_VbusOffEvent);

    if (USB_connectionInfo() & kUSB_vbusPresent)
      USB_handleVbusOnEvent();

    while(1)
    {
        switch(USB_connectionState())
        {
           case ST_ENUM_ACTIVE:
              if (!(USBCDC_intfStatus(1,&bytesTx,&bytesRx)&kUSBCDC_waitingForReceive))
                BYTE ret = USBCDC_receiveData(RXBuffer,MEGA_SIZE,1);
              .
              .
              USBCDC_intfStatus(1,&bytesTx,&bytesRx);
              if(bytesRx > threshold)
              {
                abortReceive(bytesRx,1);
                process_the_data(RXBuffer,bytesRx);
              }
              break;
        }
     }
```

Figure 21. Continuously-Open Receive

As the main loop executes, it checks to make sure a receive operation is open; if not, it opens one. Separately, it polls to see if the amount of received data has reached a certain threshold. If it has, then it processes it.

12.8.3 Fixed-Size Receive

If the amount of data to be received is known in advance, a receive operation for that exact amount can be opened at any time. When that amount has been received, a *USBxxx_handleReceiveCompleted()* event will occur. This event handler can process the data directly, or set a flag for main() to process it.

Suppose a protocol is to be implemented in which all commands from the host consist of a two-byte command code and 32 bytes of payload data. Once received, the device is to execute the command and respond with a two-byte acknowledge.

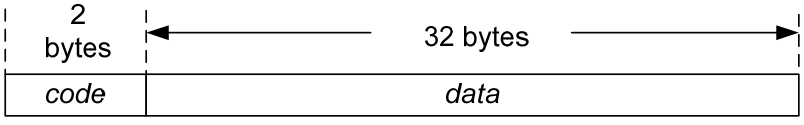


Figure 22. Command Packet

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;                  // Configure watchdog
    Init_System();
    USB_init();
    USB_setEnabledEvents(kUSB_receiveCompletedEvent);

    //Connect to USB if cable already was plugged in
    if (USB_connectionState() == USB_CONNECTED_NO_ENUM)
        USB_handleVbusOnEvent();

        while(1)
    {
      switch(USB_connectionState())
      {
        case ST_ENUM_ACTIVE:

          __disable_interrupt();
          ret = USBCDC_receiveData(command,34,0);
          if((ret == kUSBCDC_receiveStarted) || (ret == kUSBCDC_intfBusyError))
            __bis_SR_register(LPM0_bits + GIE);
          __enable_interrupt();

          // CPU was awakened from LPM0
          if(bReceiveCompleted_event)
          {
            bReceiveCompleted_event = FALSE;
            execute_the_command(command);
            if(USBCDC_receiveData(command,34,1)==kUSBCDC_busNotAvailable) //Open a new RX op
            {
              USBCDC_abortReceive(&x,1);
              break;
            }
          }
          .
          .
          default:;
        }
    }  // while(1)
} //main()


BYTE USBCDC_handleReceiveCompleted(BYTE intfNum)
{
    bReceiveCompleted_event = TRUE; // Signal that data has been received
    return TRUE;                    // Keep CPU awake after returning from event
}
```

Figure 23. Fixed-Size Receive

A receive operation is kept open at all times for 34 bytes. When fulfilled, the IF-branch clears the flag, executes the command, and opens a new receive operation.

LPM0 is entered after beginning the receive operation, but only if the return value is *kUSBCDC_receiveStarted* or *kUSBCDC_intfBusyError,* both of which would mean a receive operation is open and the application should wait in LPM0 for it to finish. If the return were *kUSBCDC_busNotAvailable,* then it means the state is no longer *ST_ENUM_ACTIVE,* and the main loop should be allowed to refresh.

Since it's possible for a receive operation to be completed between the point that the function assignts its return value and the point at which the value is evaluated in main(), and since the consequences of that return value are high (going to sleep), interrupts are disabled first and re-enabled simultaneous to entering LPM0. This way the receive operation can't begin until LPM has been entered.

13 Debugging Tips

13.1 The Device Enumeration Process

If a USB connection fails to operate as expected, a first step is to determine whether the device enumerated successfully on the host. *Enumeration* is the process by which the host identifies the device and associates it with the appropriate driver. If enumeration did not complete, then it's important to understand how far it proceeded.

13.1.1 Summary of the Enumeration Process

First, it's important to understand the enumeration process. This is shown below.

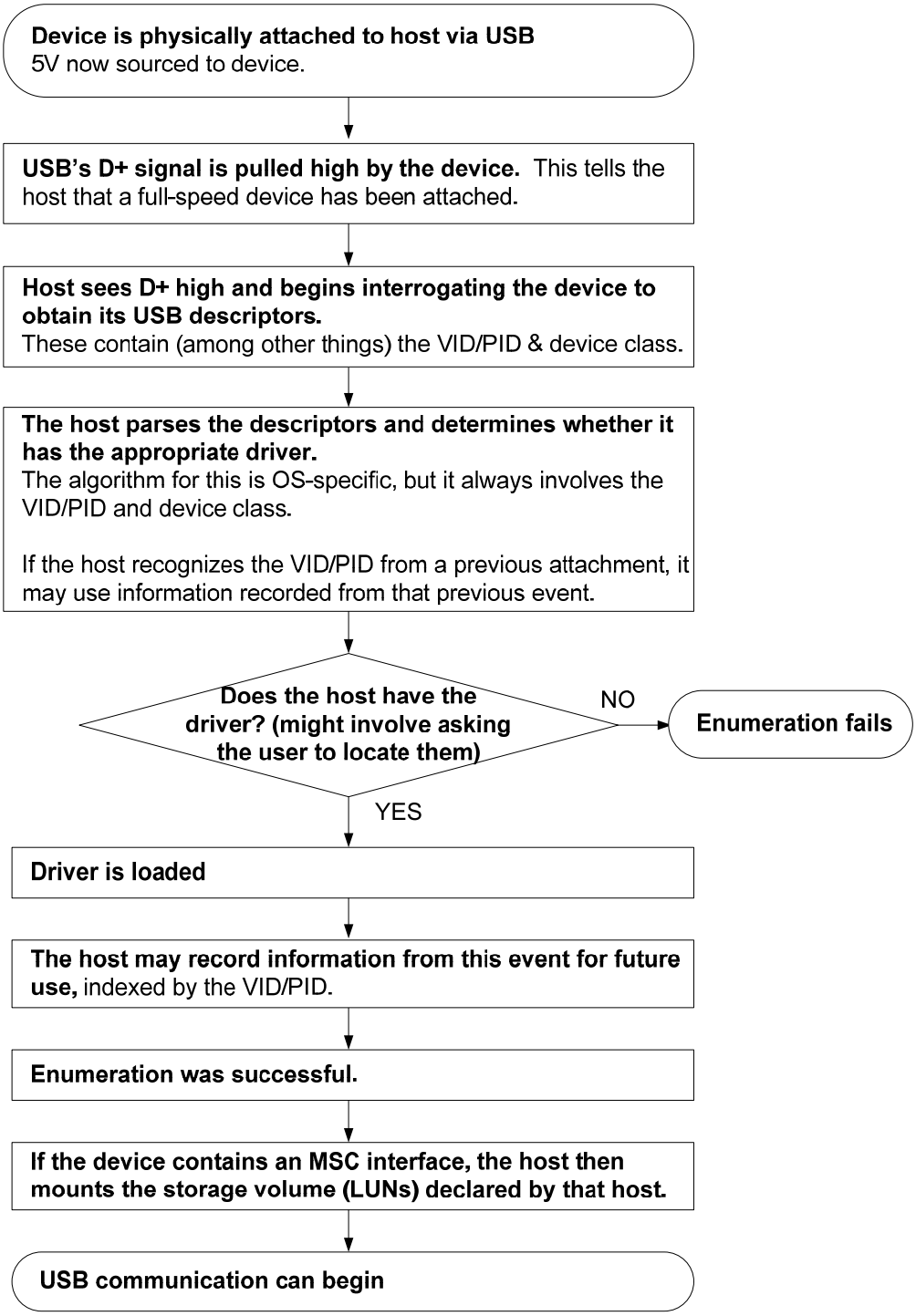


Figure 24. Basic Enumeration Flow

The first time a device is attached, enumeration will probably take longer than subsequent attachments. This is largely due to the fact that the first attachment usually involves a “device installation”, in which information about the device is recorded. (Windows does this in the system registry.) Device installation can take several seconds, even on interfaces that install silently (HID/MSC). Subsequent attachments may happen more quickly.

13.1.2 Determining Whether the Device Enumerated

Host operating systems may have ways to determine how far the enumeration process proceeded.

Windows provides various clues. The first attachment of a device may result in activity in the system tray. A CDC interface, of course, will result in a “Found New Hardware” wizard. Both of these indicate that the host at least saw the presence of the USB device (D+ was pulled high).

If the enumeration process succeeds – that is, a driver is loaded -- Windows plays audible alert tones. It can also be verified by checking the Windows Device Manager, shown in the figure below.

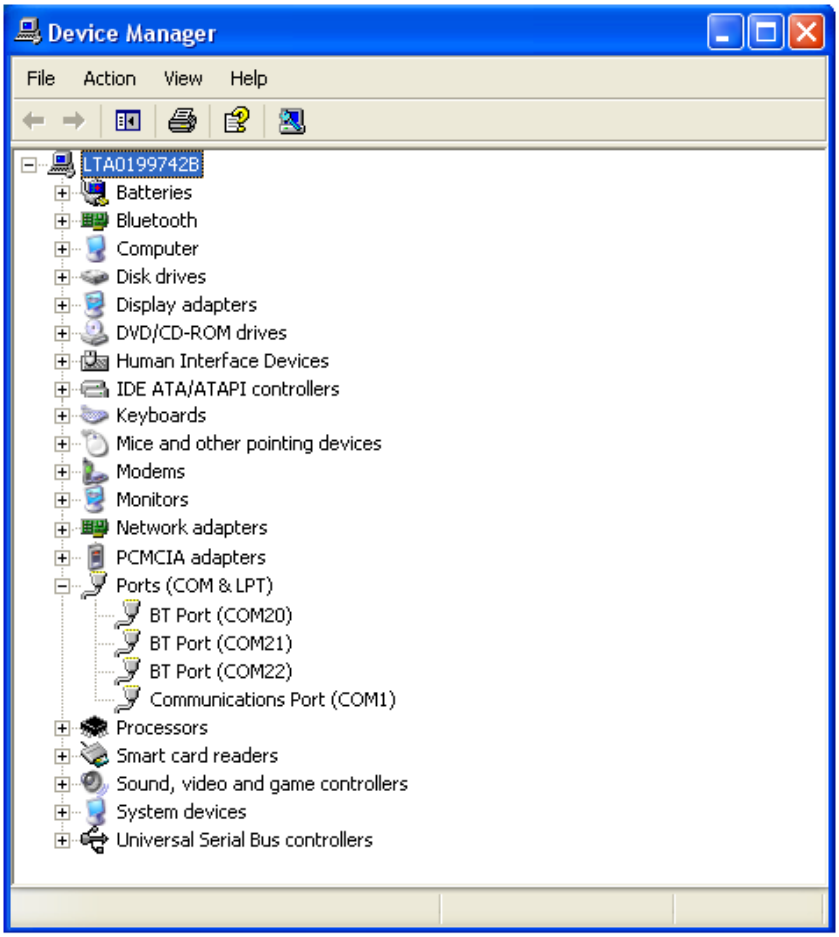


Figure 25. Windows Device Manager

The section under which the device should appear depends on the device; CDC devices enumerate under “Ports”, HID devices enumerate under “Human Interface Devices”, and MSC devices enumerate under “Disk Drives”. Whichever category it’s listed under, attaching or detaching should both cause a refresh action in the display. Selecting “Properties” for the device can reveal the VID, PID, and serial number for that device.

Other operating systems have similar ways of determining enumeration. The Mac OS doesn’t have an equivalent to the Device Manager built into the OS, but developer tools are available that provide similar information.

13.1.3 Determining if the Device Asserted Itself to the Host

If the Device Manager doesn’t refresh itself upon inserting/detaching the device, it suggests the host didn’t see the device asserting its presence (pulling D+ high). This can be confirmed with a voltmeter or oscilloscope on the PUR or D+ pins. This could be a problem with either hardware (a short to ground) or software (not calling *USB_connect()*).

13.1.4 D+ Was Asserted, but Driver Association Failed

If the device was seen by the host, but enumeration didn’t complete successfully, then somehow the OS failed to associate the device with the appropriate driver. Windows sometimes reacts to this situation by showing a “yellow bang” on this device in the Device Manager, or another negative result. This indicates the device was seen, but the driver was not loaded.

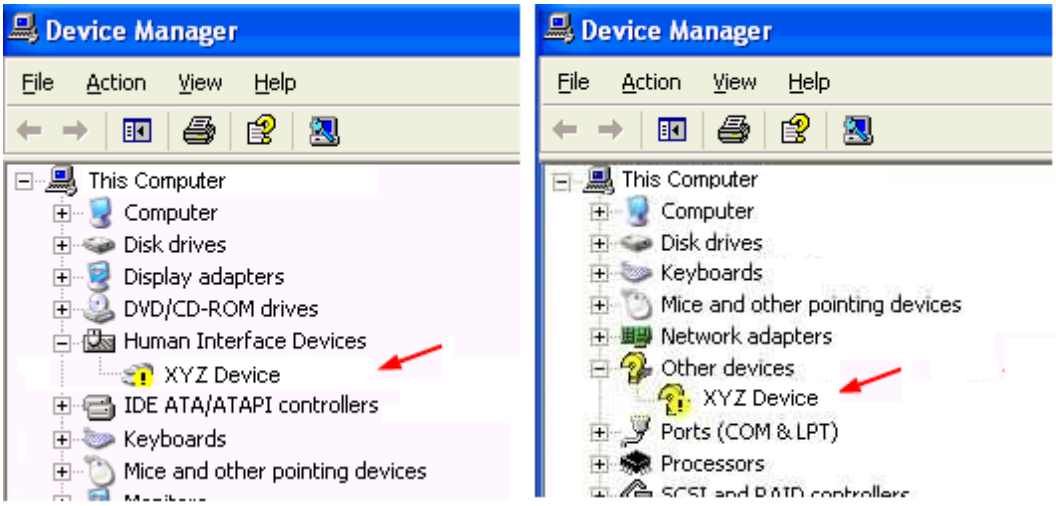


Figure 26. Failed Enumeration Results in the Device Manager

13.2 Common Problems

13.2.1 Problems that Can Cause USB Failure at Any Time

The following items are fundamental to USB operation. If any of these is violated, it can cause failure at any time during operation.

First, always check the hardware against the reference schematic located in the application note "Starting a USB Design with MSP430 MCUs" (slaa457). Missing a critical schematic element will probably cause USB failure very quickly after attempting enumeration.

Clocking is critical. If failure occurs at any time, double-check that an appropriate clock frequency is available on XT2. (See Sec. 12.2.)

Check to see if the MSP430 is experiencing any clock oscillator faults. Also check for a "bus error" NMI, which generally indicates the PLL isn't operating at an expected time (see Sec. 12.2.6).

Finally, the API can only do its job if its allowed to handle USB interrupts. If the application prevents this somehow, then USB will fail. An application might disable general interrupts directly, or it can do it indirectly by spending a lot of time in interrupt service routine handlers. (See Sec. 12.3.)

13.2.2 Problems that Can Cause Failed Enumeration (Driver Association)

The API handles enumeration automatically, but it needs a proper environment in which to do so.

First, double-check the items in the previous section, as these can cause problems in USB communication at any time.

The API performs delays after it enables the PLL and LDOs, giving them a chance to stabilize. These are simple delay loops dependent on MCLK frequency. For this reason, the Descriptor Tool's has an MCLK field (which controls the USB_MCLK_FREQ constant in *descriptors.h*). This needs to be configured for the fastest MCLK used during operation. The API adapts its delays to this value. A severe mismatch between this value and reality could cause the API to wait in inadequate amount of time when starting up the PLL and LDOs. This would probably appear as failed enumeration.

If none of these fundamental items is at fault (hardware, clocks, and power), then the problem may be occuring at the level of USB descriptors and the host's handling of them. One common problem is that a record of the device's VID/PID already exists on this host machine, but the information on record is in conflict with this device's USB descriptors. This is caused by using the same VID/PID on this host machine earlier, but with a different USB descriptor set. Therefore, when making changes in the interface set or certain descriptor fields, always either delete the previous entry, or give it a new VID/PID pair. See Sec. 13.3.

Another possibility is that the descriptors themselves contain an error. If the Descriptor Tool was used to generate the descriptors, then this shouldn't occur. It can occur, however, f the Tool's output was manually modified. After fixing the error, be sure to either delete the device's previous entry, or give this descriptor set a new VID/PID. See Sec. 13.3..

13.2.3 Problems Causing De-Enumeration Within One Minute

The most direct cause of this behavior would be if the device pulled its D+ line low, telling the host it's done communicating. Assuming this isn't the case, this behavior could occur for any reason that interferes with the fundamental requirements for USB operation listed in Sec. 13.2.

One additional cause that could lead to this specific behavioral pattern is if the device is a composite device containing an MSC interface. If the MSC interface isn't properly implemented, the host will be unable to mount the storage volume. The host may timeout waiting for the volume to mount – on the order of "seconds" – and then de-enumerate the entire device, including the non-MSC interfaces. (CDC and HID interfaces don't have a similar requirement – there are no additional requirements placed on the application, beyond those listed in the previous sections.)

The most likely offenders causing an incomplete MSC implementation would be insufficient calling of *USBMSC_poll(),* and not properly handling buffer requests for all LUNs. See Sec. 8.3 for information about proper MSC implementation.

13.3 Avoiding Device Conflicts on the Host During USB Development

As indicated above, when a USB device is enumerated for the first time, it undergoes a "device installation". The host records information from the device's USB descriptors, indexed by the device's VID/PID. The next time this host machine encounters this VID/PID, it uses this recorded information, rather than re-installing the device.

In the field, end users should never see a problem from this. USB descriptors for a given VID/PID should never change once a product is released. During development, however, the USB descriptors may change as the developer changes the device toward the final goals. It's therefore easy for the developer's host machine to see multiple descriptor sets. If the developer doesn't consciously deal with the possibility of conflicts, problems will occur on this host machine.

A simple way to deal with this is to increment the PID to a new value, each time descriptors are changed. This ensures the host machine always sees the device as a new one. Note that the INF file used with CDC interfaces also contains a VID/PID pair, and this must match the one reported by the device. As the descriptor's PID changes, so must the INF's PID. The Descriptor Tool automatically generates this file when generating the descriptor files for the USB API stacks. The files generated at a given time should be used together.

This approach is fast and easy, but a downside is that the system registry may become cluttered with new VID/PIDs. Also, it's easy to lose track of what this host machine has registered in the past. A more complete solution is to delete the VID/PID from the host machine's registry.

As an example, in Windows, this can be performed by the following procedure:

1. From the Start menu's "Run" prompt, type "regedit". This opens Windows' system registry editor.

2. Navigate to HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB

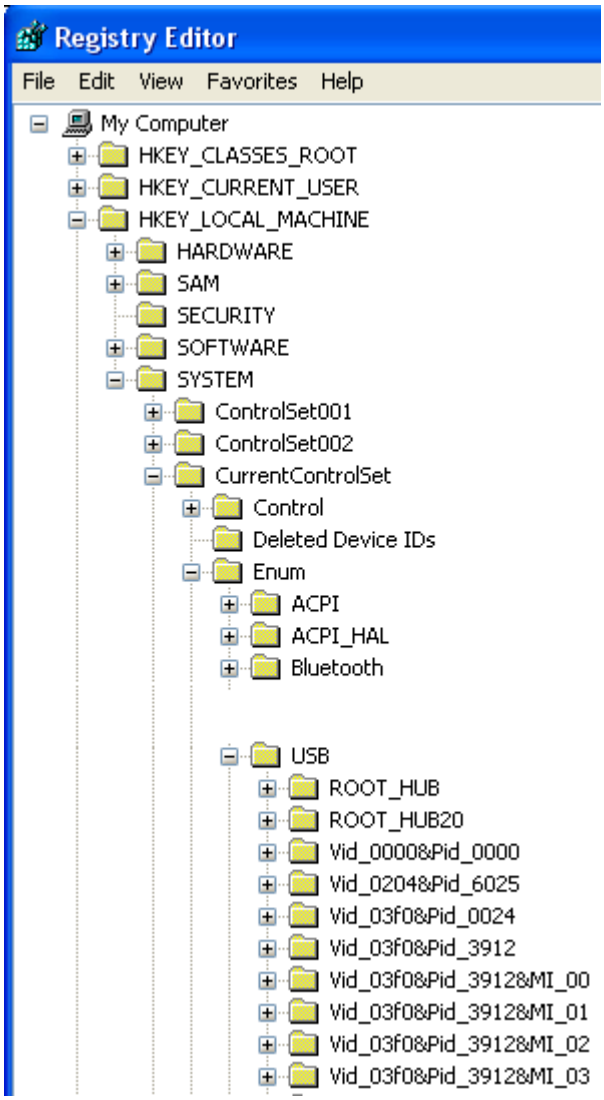


Figure 27. Locating USB Devices in RegEdit

The device's key within \USB is formed by a concatonation of VID and PID values. If the device was a composite device, then additional keys are formed for each interface, with an &MI_xx extension.

To delete the device from the registry, delete all keys containing the VID/PID being eliminated.

Sometimes, RegEdit won't allow deletion until permissions are enabled:

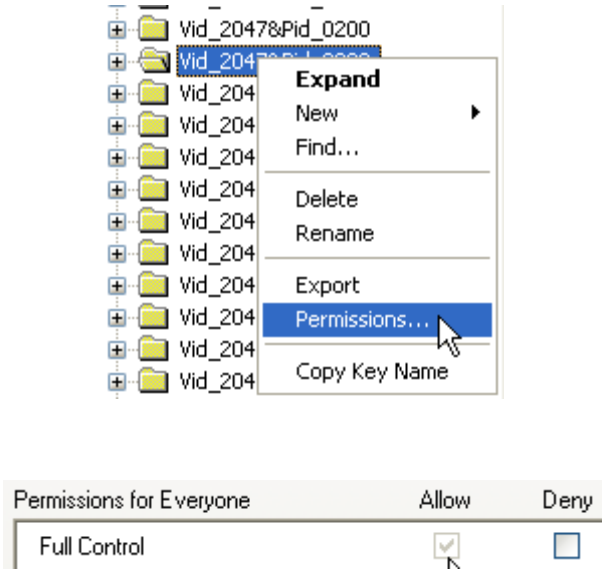


Figure 28. Enabling Permissions for a Registry Key

Click “Allow”, press the “OK” button, then try again to delete.

When the USB device is released to end users, the VID/PID must be finalized. The same applies to custom strings in the descriptors and the INF file. The Descriptor Tool can be used for this purpose.

14 Send/Receive Construct Functions for CDC and HID-Datapipe

This section serves as a reference for the example construct functions described in the section above. These functions are found in the file *USB_constructs.c.*

The discussion in this section applies to any CDC interface or HID interface using the datapipe function calls (see Sec. 7). It does not apply to traditional HID interfaces.

These functions are user-editable, if desired. An example of a modification the developer might want to make it to employ a timeout feature based on a hardware timer. These functions were written to avoid using additional hardware resource, and so the sending functions use a less elegant approach of counting the times interface status is polled. This probably works well in most situations, but a timer approach would be more accurate.

14.1 BYTE cdcSendDataWaitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)

Description

Sends the data in *dataBuf,* of size *size,* using the post-call polling method. It does so over interface *intfNum.* The function doesn't return until the send has completed. The function assumes that *size* is non-zero.

The 32-bit number *ulTimeout* selects how many times *USBCDC_intfStatus()* will be polled while waiting for the operation to complete. If the value is zero, then no timeout is employed; it will wait indefinitely. When choosing a number, it is advised to consider MCLK speed, as a faster CPU will cycle through the calls more quickly.

In many applications, the return value can simply be evaluated as zero or non-zero, where non-zero means the call failed for reasons of host or bus non-availability. Therefore, it may desirable for the application to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

It's recommended not to call this function from within an event handler. This is because if an interface currently has an open send operation, the operation will never complete during the event handler; rather, only after the ISR that spawned the event returns. Thus the *USBCDC_intfStatus()* polling would loop indefinitely (or timeout). It's better to set a flag from within the event handler, and use this flag to trigger the calling of this function from within main().

Parameters

Table 57. Parameters for *cdcSendDataWaitTilDone()*

Returns	*0:* the call succeeded; all data has been sent *1:* the call timed out, either because the host is unavailable or a COM port with an active application on the host wasn't opened. 2: the bus is unavailable.

The function is shown below, followed by commentary about how it operates.

```
// This call assumes no send operation is underway; also assumes size is non-zero.
// Returns zero if send completed; returns non-zero if main loop should exit
BYTE cdcSendDataWaitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)
{
  ULONG sendCounter = 0;
  WORD bytesSent, bytesReceived;

  switch(USBCDC_sendData(dataBuf,size,intfNum))
  {
     case kUSBCDC_sendStarted:
          break;
     case kUSBCDC_busNotAvailable:
          return 2;
     case kUSBCDC_intfBusyError:
          return 3;
     case kUSBCDC_generalError:
          return 4;
     default:;
  }

  // If execution reaches this point, then the operation successfully started.  Now
  // wait til it's finished.
  while(1)
  {
    BYTE ret = USBCDC_intfStatus(intfNum,&bytesSent,&bytesReceived);
    if(ret & kUSBCDC_busNotAvailable)               // This may happen at any time
      return 2;
    if(ret & kUSBCDC_waitingForSend)
    {
      if(ulTimeout && (sendCounter++ >= ulTimeout))  // Incr counter & try again
        return 1 ;                                  // Timed out
    }
    else
      return 0;                                     // It succeeded
  }
}
```

Figure 29. cdcSendDataWaitTilDone()

Notice this function checks every return value from *USBCDC_sendData(),* leaving nothing unhandled.

The main purpose of the timeout function is to avoid being caught here infinitely. For example, even if the bus is available, the host needs to initiate the data transfers – something it may not do if the COM port has not been opened by the host application. If this happens, and if this function didn't handle the situation, execution could poll *USBCDC_intfStatus()* indefinitely. The timeout ensures this doesn't happen. The timeout function isn't exact, but it does serve the purpose of limiting the length of retries, without using up a system timer resource.

14.2 BYTE cdcSendDataInBackground(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)

Description

Sends the data in *dataBuf,* of size *size,* using the pre-call polling method. It does so over interface *intfNum.* The send operation may still be active after the function returns, and *dataBuf* should not be edited until it can be verified that the operation has completed. The function assumes that size is non-zero.

The 32-bit number *ulTimeout* selects how many times *USBCDC_intfStatus()* will be polled while waiting for the previous operation to complete. If the value is zero, then no timeout is employed; it will wait indefinitely. When choosing a number, it is advised to consider MCLK speed, as a faster CPU will cycle through the calls more quickly.

In many applications, the return value can simply be evaluated as zero or non-zero, where non-zero means the call failed for reasons of host or bus non-availability. Therefore, it may desirable for the application to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

It's recommended not to call this function from within an event handler. This is because if an interface currently has an open send operation, the operation will never complete during the event handler; rather, only after the ISR that spawned the event returns. Thus the *USBCDC_intfStatus()* polling would loop indefinitely (or timeout). It's better to set a flag from within the event handler, and use this flag to trigger the calling of this function from within main().

Parameters

Table 58. Parameters for *cdcSendDataInBackground()*

Returns	*0:* the call succeeded; all data has been sent *1:* the call timed out, either because the host is unavailable or a COM port with an active application on the host wasn't opened. 2: the bus is unavailable.

The function is shown below. See the previous section for commentary about how it operates (the same comments apply).

```
// This call assumes a send operation might be underway; also assumes size is non-zero.
// Returns 0 if send completed; returns non-zero if main loop should exit
BYTE cdcSendDataInBackground(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)
{
  ULONG sendCounter = 0;
  WORD bytesSent, bytesReceived;

  while(USBCDC_intfStatus(intfNum,&bytesSent,&bytesReceived) & kUSBCDC_waitingForSend)
  {
    if(ulTimeout && ((sendCounter++)>ulTimeout))  // A send operation is underway;
                                                  // incr counter & try again
      return 1;                                   // Timed out
  }

  // The interface is now clear.  Call sendData().
  switch(USBCDC_sendData(dataBuf,size,intfNum))
  {
     case kUSBCDC_sendStarted:
          return 0;
     case kUSBCDC_busNotAvailable:
          return 2;
     default:
          return 4;
  }
}
```

Figure 30. cdcSendDataInBackground()

14.3 WORD cdcReceiveDataInBuffer(BYTE* dataBuf, WORD size, BYTE intfNum)

Description

Opens a brief receive operation for any data that has already been received into the USB buffer over interface *intfNum*. The data in the USB buffer is copied into *dataBuf,* and the function returns the number of bytes received.

If *size* bytes are received, the function ends, returning *size*. In this case, it's possible that more bytes are still in the USB buffer; it might be a good idea to open another receive operation to retrieve them. For this reason, operation is simplified by using large *size* values, since it helps ensure all the data is retrieved at one time.

This function is usually called when a *USBCDC_handleDataReceived()* event flags the application that data has been received into the USB buffer.

Parameters

Table 59. Parameters for cdcR*eceiveDataInBuffer*()

Returns	The number of bytes received into *dataBuf*

The function is shown below, followed by commentary about how it operates.

```
// This call assumes a receive operation is NOT underway.  It only retrieves what data
// is waiting in the buffer.  It doesn't check for kUSBCDC_busNotAvailable, because it
// doesn't matter if the bus is there or not.  size is the maximum that is allowed to be
// received before exiting; i.e., it is the size allotted to dataBuf.
WORD cdcReceiveDataInBuffer(BYTE* dataBuf, WORD size, BYTE intfNum)
{
  WORD bytesInBuf;
  BYTE* currentPos=dataBuf;

  while(bytesInBuf = USBCDC_bytesInUSBBuffer(intfNum)) // # of bytes already in USB buffer
  {
    if((WORD)(currentPos-dataBuf+bytesInBuf) <= size)
      rxCount = bytesInBuf;
    else rxCount = size;

    USBCDC_receiveData(currentPos,bytesInBuf,intfNum); // Obtain the bytes waiting
    currentPos += bytesInBuf;
  }
  return (currentPos-dataBuf);                        // Return the total bytes received
}
```

Figure 31. cdcReceiveDataInBuffer()

The function ignores the return from *USBCDC_receiveData(),* because it already knows the answer will be *kUSBCDC_receiveCompleted.* It knows this because the bytes are already in the USB endpoint buffer; it doesn't matter of the bus has been disconnected, and none of the other return codes are possible.

USBCDC_bytesInUSBBuffer() is called repeatedly, in case more data arrived while the first data was retrieved. This situation could repeat indefinitely, so it continues to poll until *USBCDC_bytesInUSBBuffer()* returns zero. This will happen when the stream of host data stops.

14.4 BYTE hidSendDataWaitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)

Description

Sends the data in *dataBuf,* of size *size,* using the post-call polling method. It does so over interface *intfNum.* The function doesn't return until the send has completed. The function assumes that *size* is non-zero.

The 32-bit number *ulTimeout* selects how many times *USBHID_intfStatus()* will be polled while waiting for the operation to complete. If the value is zero, then no timeout is employed; it will wait indefinitely. When choosing a number, it is advised to consider MCLK speed, as a faster CPU will cycle through the calls more quickly.

In many applications, the return value can simply be evaluated as zero or non-zero, where non-zero means the call failed for reasons of host or bus non-availability. Therefore, it may desirable for the application to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

It's recommended not to call this function from within an event handler. This is because if an interface currently has an open send operation, the operation will never complete during the event handler; rather, only after the ISR that spawned the event returns. Thus the *USBHID_intfStatus()* polling would loop indefinitely (or timeout). It's better to set a flag from within the event handler, and use this flag to trigger the calling of this function from within main().

Parameters

Table 60. Parameters for *hidSendDataWaitTilDone()*

Returns	*0:* the call succeeded; all data has been sent *1:* the call timed out, either because the host is unavailable or a COM port with an active application on the host wasn't opened. 2: the bus is unavailable.

The function is shown below, followed by commentary about how it operates.

```
// This call assumes no send operation is underway; also assumes size is non-zero.
// Returns zero if send completed; returns non-zero if main loop should exit
BYTE hidSendDataWaitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)
{
  ULONG sendCounter = 0;
  WORD bytesSent, bytesReceived;

  switch(USBHID_sendData(dataBuf,size,intfNum))
  {
     case kUSBHID_sendStarted:
          break;
     case kUSBHID_busNotAvailable:
          return 2;
     case kUSBHID_intfBusyError:
          return 3;
     case kUSBHID_generalError:
          return 4;
     default:;
  }

  // If execution reaches this point, then the operation successfully started.  Now
wait til it's finished.
  while(1)
  {
    BYTE ret = USBHID_intfStatus(intfNum,&bytesSent,&bytesReceived);
    if(ret & kUSBHID_busNotAvailable)               // This may happen at any time
      return 2;
    if(ret & kUSBHID_waitingForSend)
    {
      if(ulTimeout && (sendCounter++ >= ulTimeout))  // Incr counter & try again
        return 1 ;                                  // Timed out
    }
    else
      return 0;                                     // It succeeded
  }
}
```

Figure 32. hidSendDataWaitTilDone()

Notice this function checks every return value from *USBHID_sendData(),* leaving nothing unhandled.

The main purpose of the timeout function is to avoid being caught here infinitely. For example, even if the bus is available, the host needs to initiate the data transfers – something it may not do if the COM port has not been opened by the host application. If this happens, and if this function didn't handle the situation, execution could poll *USBHID_intfStatus()* indefinitely. The timeout ensures this doesn't happen. The timeout function isn't exact, but it does serve the purpose of limiting the length of retries, without using up a system timer resource.

14.5 BYTE hidSendDataInBackground(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)

Description

Sends the data in *dataBuf,* of size *size,* using the pre-call polling method. It does so over interface *intfNum.* The send operation may still be active after the function returns, and *dataBuf* should not be edited until it can be verified that the operation has completed. The function assumes that size is non-zero.

The 32-bit number *ulTimeout* selects how many times *USBHID_intfStatus()* will be polled while waiting for the previous operation to complete. If the value is zero, then no timeout is employed; it will wait indefinitely. When choosing a number, it is advised to consider MCLK speed, as a faster CPU will cycle through the calls more quickly.

In many applications, the return value can simply be evaluated as zero or non-zero, where non-zero means the call failed for reasons of host or bus non-availability. Therefore, it may desirable for the application to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

It's recommended not to call this function from within an event handler. This is because if an interface currently has an open send operation, the operation will never complete during the event handler; rather, only after the ISR that spawned the event returns. Thus the *USBHID_intfStatus()* polling would loop indefinitely (or timeout). It's better to set a flag from within the event handler, and use this flag to trigger the calling of this function from within main().

Parameters

Table 61. Parameters for *hidSendDataInBackground()*

Returns	*0:* the call succeeded; all data has been sent *1:* the call timed out, either because the host is unavailable or a COM port with an active application on the host wasn't opened. 2: the bus is unavailable.

The function is shown below. See the previous section for commentary about how it operates (the same comments apply).

```
// This call assumes a send operation might be underway; also assumes size is non-zero.
// Returns 0 if send completed; returns non-zero if main loop should exit
BYTE hidSendDataInBackground(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)
{
  ULONG sendCounter = 0;
  WORD bytesSent, bytesReceived;

  while(USBHID_intfStatus(intfNum,&bytesSent,&bytesReceived) & kUSBHID_waitingForSend)
  {
    if(ulTimeout && ((sendCounter++)>ulTimeout))  // A send operation is underway;
                                                  // incr counter & try again
      return 1;                                   // Timed out
  }

  // The interface is now clear.  Call sendData().
  switch(USBHID_sendData(dataBuf,size,intfNum))
  {
     case kUSBHID_sendStarted:
          return 0;
     case kUSBHID_busNotAvailable:
          return 2;
     default:
          return 4;
  }
}
```

Figure 33. hidSendDataInBackground()

14.6 WORD hidReceiveDataInBuffer(BYTE* dataBuf, WORD size, BYTE intfNum)

Description

Opens a brief receive operation for any data that has already been received into the USB buffer over interface *intfNum*. The data in the USB buffer is copied into *dataBuf,* and the function returns the number of bytes received.

If *size* bytes are received, the function ends, returning *size*. In this case, it's possible that more bytes are still in the USB buffer; it might be a good idea to open another receive operation to retrieve them. For this reason, operation is simplified by using large *size* values, since it helps ensure all the data is retrieved at one time.

This function is usually called when a *USBHID_handleDataReceived()* event flags the application that data has been received into the USB buffer.

Parameters

Table 62. Parameters for hidR*eceiveDataInBuffer*()

Returns	The number of bytes received into *dataBuf*

The function is shown below, followed by commentary about how it operates.

```
// This call assumes a receive operation is NOT underway.  It only retrieves what data
// is waiting in the buffer.  It doesn't check for kUSBHID_busNotAvailable, because it
// doesn't matter if the bus is there or not.  size is the maximum that is allowed to be
// received before exiting; i.e., it is the size allotted to dataBuf.
WORD hidReceiveDataInBuffer(BYTE* dataBuf, WORD size, BYTE intfNum)
{
  WORD bytesInBuf;
  BYTE* currentPos=dataBuf;

  while(bytesInBuf = USBHID_bytesInUSBBuffer(intfNum)) // # of bytes already in USB buffer
  {
    if((WORD)(currentPos-dataBuf+bytesInBuf) <= size)
      rxCount = bytesInBuf;
    else rxCount = size;

    USBHID_receiveData(currentPos,bytesInBuf,intfNum); // Obtain the bytes waiting
    currentPos += bytesInBuf;
  }
  return (currentPos-dataBuf);                        // Return the total bytes received
}
```

Figure 34. hidReceiveDataInBuffer()

The function ignores the return from *USBHID_receiveData(),* because it already knows the answer will be *kUSBHID_receiveCompleted.* It knows this because the bytes are already in the USB endpoint buffer; it doesn't matter of the bus has been disconnected, and none of the other return codes are possible.

USBHID_bytesInUSBBuffer() is called repeatedly, in case more data arrived while the first data was retrieved. This situation could repeat indefinitely, so it continues to poll until *USBHID_bytesInUSBBuffer()* returns zero. This will happen when the stream of host data stops.

15 MSP430 USB API Application Examples

Several application examples are provided with the API stack software, to demonstrate usage of the the API. Each is described here.

Most of the examples are very simple, in order to minimize the hardware requirements.

Table 63. MSP430 USB API Application Examples, and PID Map

Number	Name	Interface Type	USB VID/PID
C1	Command-line interface with LED on/off/flash	CDC	0x2047 / 0x0300
C2	Receive 1K data		
C3	Echo back to host		
C4	Packet protocol		
C5	High-bandwidth sending using *cdcSendDataWaitTilDone()*		
C6	Efficient sending using *cdcSendDataInBackground()*		
H1	Command-line interface with LED on/off/flash	HID-Datapipe	0x2047 / 0x0301
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	SD-card		0x2047 / 0x0317
M3	Multiple-LUN		0x2047 / 0x0318
CH1	Composite CDC+HID Device; communicate between Terminal and HID Demo App	CDC+HID	0x2047 / 0x0302
CC1	Composite CDC+CDC Device; communicate between two terminal apps	CDC+CDC	0x2047 / 0x0313
HH1	Composite HID+ HID Device; communicate between two HID Demo App Instances	HID+HID	0x2047 / 0x0314
CHM1	Composite CDC+HID+MSC, in which the MSC consists of two LUNs	CDC+HID+MSC	0x2047 / 0x0319
User experimentation area			0x2047 / 0x03DF - 0x03FF

15.1 The Examples' VIDs/PIDs

Sec. 13.3 explained how device conflicts can occur on a host machine during USB development, and the importance of ensuring that when descriptors change, the VID/PID do also.

With this in mind, consider the example table above. In each case, the VID is the one owned by TI MSP430: 0x2047. In some cases, the PIDs are shared by multiple examples; these are examples that all share the same descriptor set. This is true for the single-interface CDC and single-interface HID-Datapipe examples.

In other cases, the PID is unique. 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 media, which is different between the examples.

If there is any doubt, it is always safest to provide a unique VID/PID to the host.

For the case that the developer wishes to experiment with new interface sets, report descriptors, etc., TI-MSP430 has provided the "experimentation area" shown in the table – a set of 32 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 is no risk that these PIDs have been encountered by any host machine unless its owner/developer was the one who caused it.

15.2 General Instructions to Run the Examples

As discussed in Sec. 4.1.3, the examples are designed to run, "out-of-the-box", on any USB-equipped MSP430 derivative, provided a simple configuration change is made (which will be described below).

15.2.1 Hardware

Most of the examples were developed to run on the FET target boards – for example, the TS430PN80USB for the MSP430F5529. These boards can be obtained by going to the product page on http://www.ti.com for the desired device. (Do a device number search on the TI homepage, and then do a page search for the word "target".)

One of the MSC examples (M2) was designed to run on the F5529 Experimenter's Board. To obtain this board, follow the links from the F5529 product page.

If running on different hardware, there are a few items to consider:

- Some of the application examples assume the presence of an LED or button or a certain I/O pin located on the corresponding FET target board. If other hardware is used, this pin setting may need to be modified in software for the example to run properly.
- Ensure that the XT2 crystal frequency is configured properly. Since the example applications set the frequency based on the USB_XT_FREQ value, one way to do this is to load the Descriptor Tool *.dat file provided with the example, modify the XT2 frequency; re-generate the output; and use the new files to replace the default ones in the project directory.

Or, directly modify USB_XT_FREQ in *descriptors.h.* (If doing this, check the MSP430 header file (i.e., msp430f5529.h) for available values for "USBPLL_SETCLK_xxxx".)

- The port I/O configuration in the function *Init_Ports()* may need to be adjusted. The examples initialize ports with assumptions about how the I/Os are connected. They're intended to eliminate floating inputs, so they configure some I/Os as outputs. This could interfere with custom hardware. A quick way to deal with this might be to configure them as inputs, although keep in mind power draw will increase this way.

15.2.2 Loading the Example Projects with IAR and CCS

For each example, projects are provided for both the IAR and CCS environments. Be sure to use the most recent version, available for download from http://www.ti.com/msp430.

Please note that the MSC examples do not run on IAR's Kickstart (free, code-size-limited version). The code size exceeds the 8K object code limit. The free version of CCS has a higher limit (16K), and thus can be used instead.

In addition to source code and IAR/CCS projects, each example's directory includes two additional files:

- A *.dat file, which is the source input file format for the Descriptor Tool. This file was used to generate the *descriptors.c/h* and *usbisr.c* included in this example.
- If the example contains a CDC interface, an *.inf file is included. It is placed in the \USB_config subdirectory of each example, alongside the other files output by the Descriptor Tool. Windows will ask for this file when enumerating the device.

To use the examples with IAR, open the workspace file (*.eww) for the desired group of examples. 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:

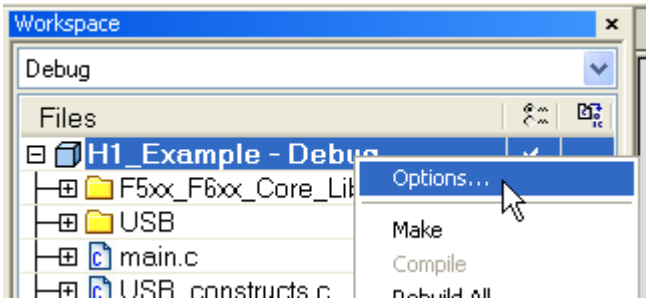


Figure 35. Opening "Project Options" in IAR

In the view shown below, choose the MSP430 device derivative being used:

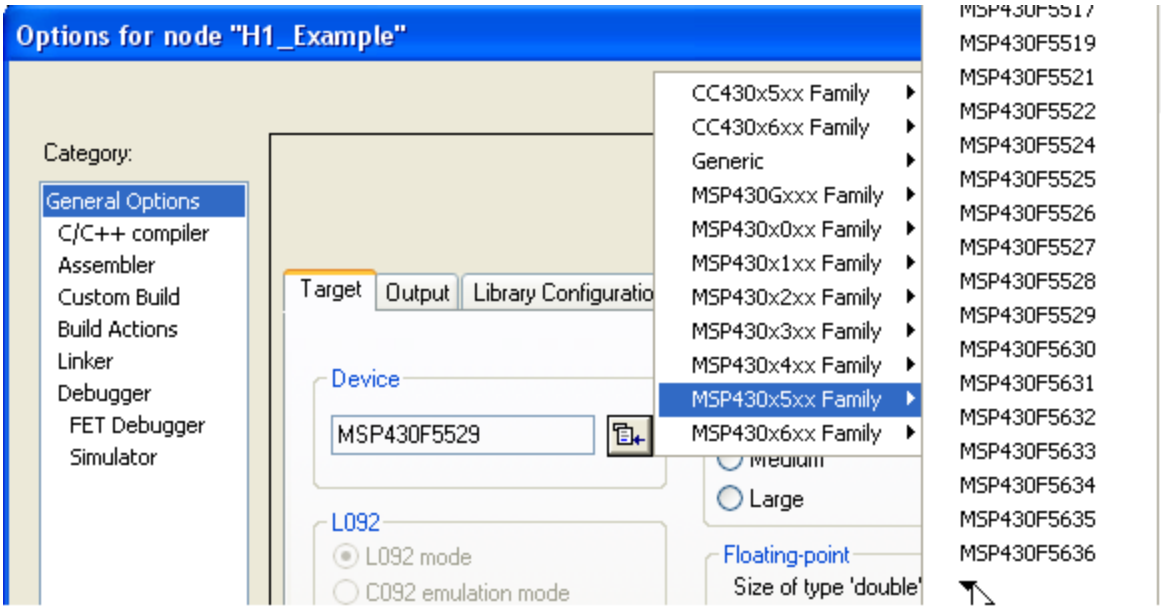


Figure 36. Choosing the MSP430 Device Derivative in IAR

Then click “OK”. The project is now configured for the appropriate device. IAR automatically predefines a symbol based on this device, and the examples detect this and adapt accordingly, using compiler directives. (The API code itself only changes its selection of MSP430 header files, but at the application level, some minor changes are made in the intialization of clcoks and ports.)

Because of the need to remove some of the IAR project’s debugger files prior to distribution, the project by default will have the debugger set to “simulator”. (Project Options → Debugger) When opening the example projects for the first time, it will always be necessary to set this back to “FET Debugger” prior to downloading to the target MSP430; otherwise code won’t be downloaded into the MSP430, but will enter the simulator instead.

Using the examples with CCS is similar. Open CCS, directing the workspace to the example group’s directory containing the “.metadata” directory (not the “.metadata” file itself).

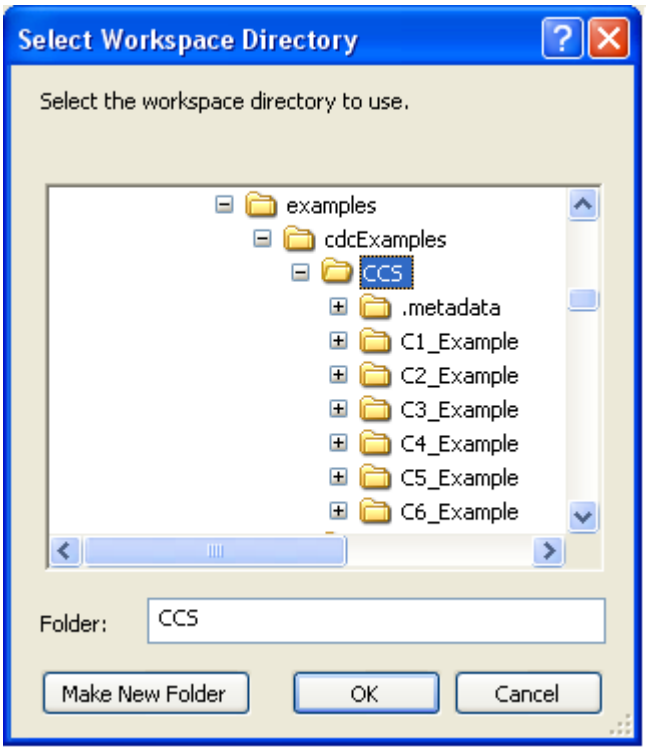


Figure 37. Selecting the Appropriate CCS Workspace Directory

When CCS finishes loading, it should show a workspace view containing all the examples in the selected group. As with IAR, one of the projects in the "workspace" view should be highlighted in bold; this is the active project. (If a different example is desired, right-click on it and select "set as active project".)

To select the device being used, open the project's properties as shown below.

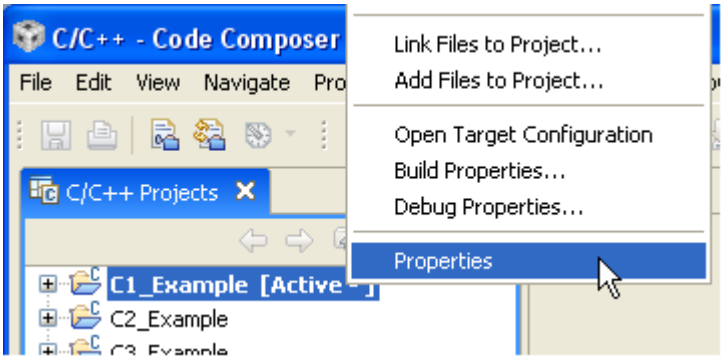


Figure 38. Opening the Project's "Properties" in CCS

Then select the “CCS Build” view, and select the “Device Variant”, as shown in red below, and press “OK”.

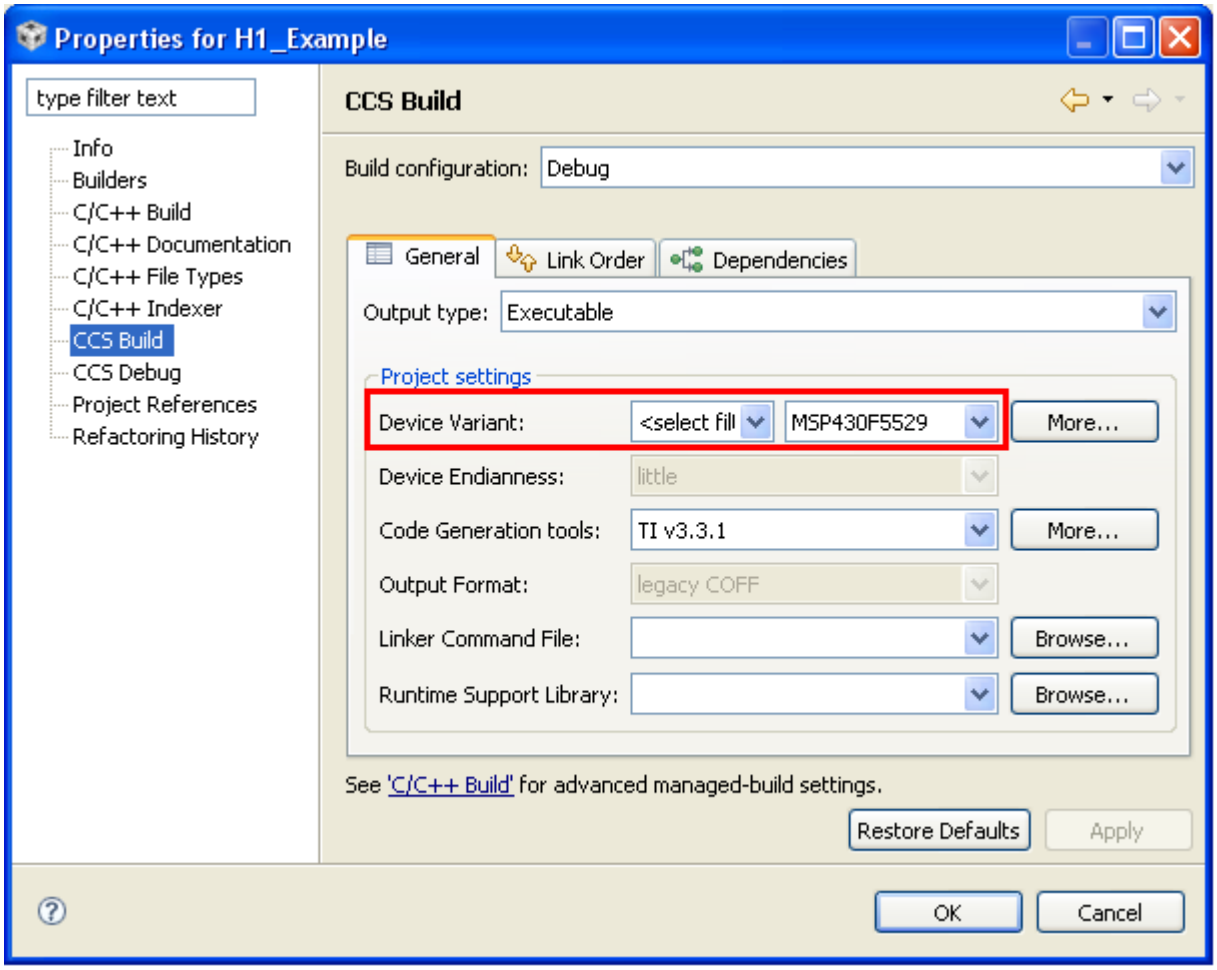


Figure 39. Choosing the MSP430 Device Derivative in CCS

The project is now fully configured for the new device. Like with IAR, the API and application adapt to the device selection.

In either environment, if compiling for a device that isn’t the “flagship” of its family (that is, any device other than F5529, F5510, F5638, or F6638), there may be compiler errors in the function *Init_Ports()*. This function initializes every port register, in order to eliminate any floating inputs. (Floating inputs cause extra current to be drawn through DVCC. See the *F5xx Family User’s Guide* for more information.) Smaller devices within a family don’t always have every I/O pin the flagship does, and thus compiling for these devices may generate errors for attemping to initialize a non-existent register. If this occurs, simply comment-out the offending lines. The example will still work correctly from a USB standpoint. However, if making current measurements, be sure that the port configuration exactly matches the I/Os available on the device, to ensure no floating inputs exist.

15.2.3 Host Software

15.2.3.1 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, like Hyperterminal. Any terminal application can be used, but behavior can differ between terminal applications. Therefore, Hyperterminal is recommended for best operation.

15.2.3.2 Datapipe HID Interfaces

TI provides a utility for HID called the HID Demo App (*HID Demo App.exe*). This utility is the compiled output of the demo application for the *Windows HID API.* Its functionality is very similar to that of terminal applications, except it uses HID as the underlying interface. It’s designed specifically for use with the HID-Datapipe function calls. This utility can be found with the HID examples.

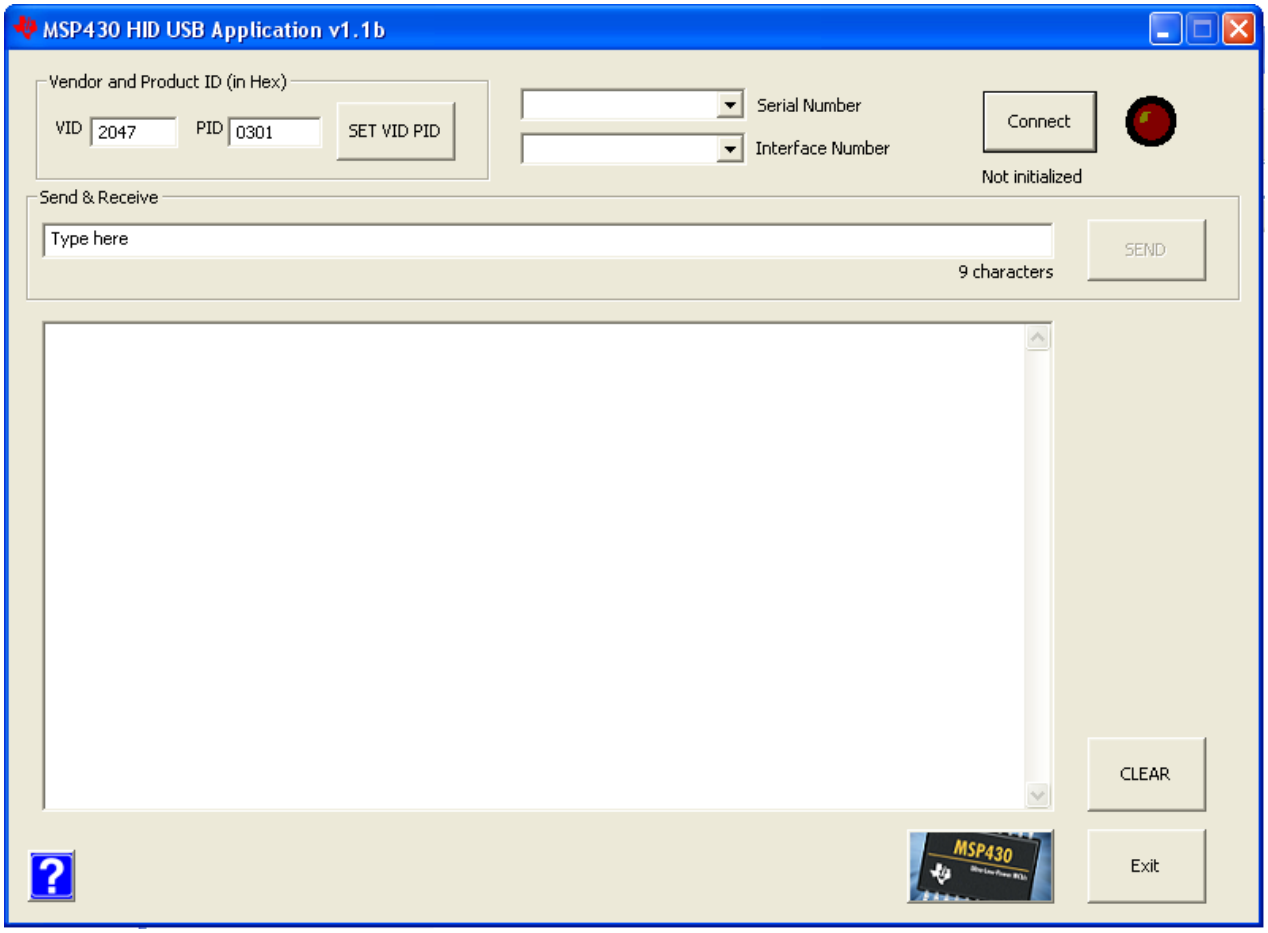


Figure 40. HID Demo AppWindow

The application is capable of locating any USB interface attached to it. Using a VID/PID, it narrows its search to only devices of a particular type. It then lists the serial numbers of all devices attached to it with that VID/PID. Each serial number represents a different physical USB device. Finally, within a physical device, it lists all the HID interfaces present, in the order they're defined in the USB descriptors ("HID 0", "HID 1", etc.). Once all of these have been selected, press the "Connect" button.

If connection is not successful, try the following:

- Check the Windows Device Manager to ensure that the device successfully enumerated on the system as an HID device
- Ensure that the VID/PID shown in *descriptors.h* matches the value entered into the VID/PID fields in the utility
- 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.

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

15.2.3.3 Traditional HID Interfaces

These examples use Windows as the "application", by reporting themselves to be a mouse or keyboard.

15.2.3.4 MSC Interfaces

These examples use Windows as the "application". Interact with these devices using the host's usual mechanisms for storage volumes. Some of the examples specfically use text files, for interaction with simple text editors (i.e., Windows' "Notepad").

15.2.4 CDC Interfaces: INF Files and Device Installation on Windows

All the examples include an INF file in their \USB_config subdirectory. When attaching a CDC device to a Windows machine, Windows must be able to find an INF that allows it to associate the device with the correct driver. When dealing with multiple examples and enumerations, as the developer is, it's very beneficial to understand how Windows deals with INF files. (Fortunately the end user only needs to go through a single device installation, and so for them it is simple; but the developer might do this many times.)

Every Windows CDC INF file is associated with two unique aspects:

- A unique VID/PID pair
- A unique set of USB interfaces (which itself should be associated with a unique VID/PID pair)

The first time enumerating a new device (new combination of VID, PID, and serial number), Windows performs a device installation process. In Windows XP, it begins like this:

Figure 41. Windows' Device Installation ("Found New Hardware") Box

If this was a product on the market, and if the driver had been previously certified with Microsoft, the end user might be able to use Windows Update. This would be the easiest process. Since that isn't the case for the engineer developing a product, the engineer should select "No, not this time". This causes the following box to display:

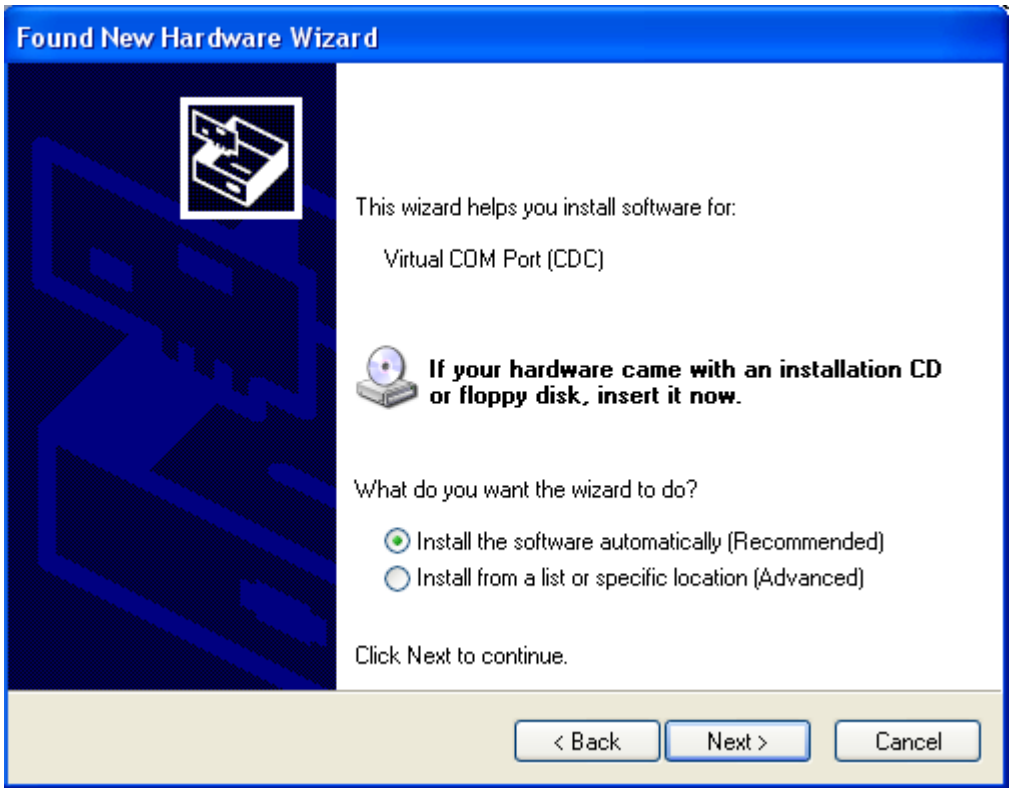


Figure 42. Device Installation, Second Box

If this box is displayed, and if the user believes this same INF has been used on this machine before, choose the first item “Recommended”. Windows will look in its history and find it. If it doesn’t find it, this will fail and bring the user back to the box shown above.

If the INF hasn’t been used on this machine before, select “Advanced”, then press “Browse”. This opens a “file open” box. Guide it to the 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.)

After a successful device installation, when enumerating any other example 1-6 – all of which have the same VID/PID and, if using the same board each time, same serial number – Windows will see it as the same device each time, and not prompt a device installation process.

If instead, one of the examples is subsequently run from a different piece of hardware, Windows will recognize the VID/PID but still prompt a device installation process. This time, it is possible to select the “Recommended” button; Windows will look in the system registry and find the necessary information.

When running the composite examples for the first time, Windows will see a new VID/PID. These examples are loaded with unique PIDs because their USB descriptors are different. Among other things, this is necessary because Windows stores descriptor information in the system registry, using the VID/PID as an index. Windows needs a different INF file to load these, so it’s important to guide Windows to the INF files located in the directories for those examples.

All of this can be a little complicated. However, 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.

15.2.5 General Instructions for Running Examples on a Windows PC

For each example, open the IAR/CCS workspace, download the project, and run it. Attach the hardware to a PC. If the device contains a CDC interface, Windows will prompt for a device installation, as discussed in the previous section. Follow the instructions in that section to complete the process.

If the interface is CDC, then start Hyperterminal and select the appropriate COM port. (After an installation, this is usually the one with the highest number. If necessary, use the Windows Device Manager to identify which COM port is associated with the MSP430.) The baudrate and other COM port configuration settings do not matter.

If the interface is HID, then start the HID Demo App. Set the VID to 0x2047 and the PID to 0x0301 (unless otherwise indicated in the example's instructions), and press "Set VID/PID". A value should appear in the "Serial Number" and "Interface Number" menus. Press the "Connect" button to initiate a connection with the HID interface.

When working with terminal applications like Hyperterminal, it will soon be noticed that they're not very tolerant of the COM port disappearing during operation. Because of this, the terminal application must close and re-open the port whenever the MSP430 program is reset while the port is open, or if the cable is removed, before attempting to re-connect. There's an order in which this must be done:

- Close the terminal's connection to the port
- Connect the USB device to the PC; if the volume is up, Windows' alert can be heard indicating that a new device has been attached
- Now, re-open the terminal's connection to the port

In short, the port should not be already open while the device enumerates. This is simply a symptom of using an old software mechanism (COM ports) over a new, dynamic medium (USB)

15.3 Example #C1: Single-CDC; Command-Line Interface with LED On/Off/Flash

15.3.1 Running It

This example implements a simple command-line interface, where the command is ended by pressing 'return'. It accepts four "commands":

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

These commands control the LED on an MSP430 FET board. If using other hardware that has an LED attached to an MSP430 I/O pin, the code can be easily changed to use that I/O.

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.

15.3.2 Implementation Comments

This example uses the *receiveDataInBuffer()* construct function described in Sec. 12.8.1. This function is advantageous in this application, because with a return-delimited command interface, there's no way to predict the exact number of bytes 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 *cdcSendDataWaitTilDone()* to send the response data. *cdcSendDataInBackground()* could have been used as well, with no practical difference in the application.

15.4 Example #C2: Single-CDC; Receive 1K Data

15.4.1 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. This can be sent from Hyperterminal using the "Send text file" command. (Note that Hyperterminal and many other terminal applications use 1-byte packets for this function, and therefore the transfer is slow.)

15.4.2 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* begins 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.

15.5 Example #C3: Single-CDC; Echo Back to Host

15.5.1 Running It

This application simply echoes back characters it receives from the host. Unless the terminal application has a built-in echo feature turned on, 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.

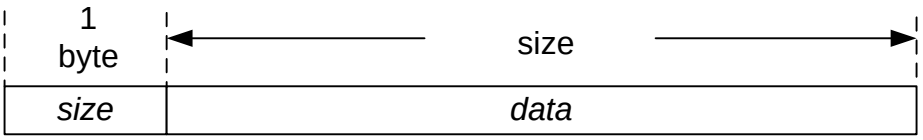
15.5.2 Implementation Comments

Since there's no predicting 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.

15.6 Example #C4: Single-CDC; Packet Protocol

15.6.1 Running It

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



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.

15.6.2 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.

15.7 Example #C5: Single-CDC; High-Bandwidth Sending Using *cdcSendDataWaitTilDone()*

15.7.1 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.

15.7.2 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, discussed in Sec. 4.2.3. If using Hyperterminal, bandwidth will be slow because it uses only a single byte per USB packet.

15.8 Example #C6: Single-CDC; Efficient Sending Using *cdcSendDataInBackground()*

15.8.1 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.

It's recommended to review Sec. 7.2.2, which discuss the API's background processing.

15.8.2 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 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 *cdcSendDataWaitTilDone().*

15.9 Example #H1: Single-HID; Command-Line Interface with LED On/Off/Flash

15.9.1 Running It

This example implements a simple command-line interface, where the command is ended by pressing 'return'. It accepts four "commands":

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

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

These commands, entered with the HID Demo App, control the LED on an MSP430 FET board. If using other hardware that has an LED attached to an MSP430 I/O pin, the code can be easily changed to use that I/O.

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.

15.9.2 Implementation Comments

This example uses the *hidReceiveDataInBuffer()* construct function described in Sec. 12.8.1. This function is advantageous in this application, because with a return-delimited command interface, there's no way to predict the exact number of bytes 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 *hidSendDataWaitTilDone()* to send the response data. *hidSendDataInBackground()* could have been used as well, with no practical difference in the application.

15.10 Example #H2: Single-HID; Receive 1K Data

15.10.1 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.

15.10.2 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.

15.11 Example #H3: Single-HID; Echo Back to Host

15.11.1 Running It

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

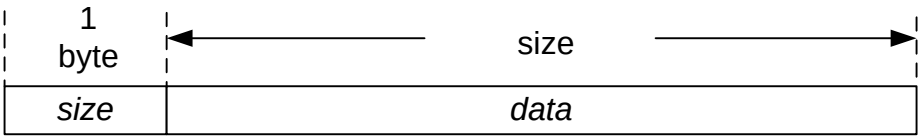
15.11.2 Implementation Comments

Since there's no predicting 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.

15.12 Example #H4: Single-HID; Packet Protocol

15.12.1 Running It

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



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 packet by itself, with no data behind it. It won't work to send "3abc" in a single transmission.

15.12.2 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.

15.13 Example #H5: Single-HID; High-Bandwidth Sending Using *hidSendDataWaitTilDone()*

15.13.1 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.

15.13.2 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 timeouts 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.

15.14 Example #H6: Single-HID; Efficient Sending Using *hidSendDataInBackground()*

15.14.1 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.

It's recommended to review Sec. 7.2.2, which discuss the API's background processing.

15.14.2 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().*

15.15 Example #H7: Traditional Single-HID (Mouse); Sending/Receiving Reports

15.15.1 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. In HID-Traditional applications, the host operating system acts as the "application", rather than a user-created application running on that host. Further, the HID Demo App is designed for HID-Datapipe communication, not for custom HID reports.

15.15.2 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.* Windows recognizes certain things about this report 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 FET target boards, which don'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.

With HID-Traditional function calls, the software designer is responsible for everything related to the report format. This includes creating the report descriptor in *descriptors.c* (must be pasted in after the Descriptor Tool is run), and it also includes building reports that match the report descriptor. With HID-Datapipe, these things aren't necessary.

15.16 Example #H8: Traditional Single-HID (Keyboard); Sending/Receiving Reports

15.16.1 Running It

This example functions as a keyboard on the host. It's designed to run on a FET target board. Once enumerated, pressing one of the target board's 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$#)".

Unlike the HID-Datapipe examples, this one does not communicate with the HID Demo Application. In HID-Traditional applications, the host operating system acts as the "application", rather than a user-created application running on that host. Further, the HID Demo App is designed for HID-Datapipe communication, not for custom HID reports.

15.16.2 Implementation Comments

A custom report descriptor is defined in *descriptors.c.* Windows recognizes certain things about this report 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 mouse data for the pointer.

With HID-Traditional function calls, the software designer is responsible for everything related to the report format. This includes creating the report descriptor in *descriptors.c* (must be pasted in after the Descriptor Tool is run), and it also includes building reports that match the report descriptor.

15.17 Example #M1: Single-LUN; File System Emulation

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 ten blocks (512 x 10 = 5K) are emulated, including 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 a single file is supported (although it could probably be adapted to 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.

15.17.1 Running It

This example shows a data logging application like the one described above. It is specifically designed to operate on a FET target board, but can also work with other hardware.

Since part of the functionality is the logging of data when not attached to USB, be sure to power the board from a non-USB source (such as the FET programming tool). Build and run the example, and without enumerating the device, press the S1 button several times. Each press simulates another entry of "data" into the data log file.

Then enumerate the device. 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. Several entries should be found there, one for each button press. Changes can be made and saved from Notepad as well.

If you don't press S1 but rather simply enumerate the device, some default text appears in the file.

15.17.2 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 UsbMscUserData.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).

Finally, the data block itself (Data[]) can be modified. The data logging code in main() modifies this block.

The Master Boot Record (MBR[]),file allocation table (FAT[]), and root directory (Root_Dir[]) are stored in flash. Only the data block (Data[]) is stored in RAM. 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. (Since the volume data indicates two blocks per cluster, a second data block could be added, i.e. Data1[] and Data2[], without having to adjust the FAT.)

15.18 Example #M2: SD-Card Reader

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

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.

When run on the F5529 Experimenter's Board, this example implements a fully-functional SD-Card reader. It detects (and tolerates) 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.

15.18.1 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. (Future versions of the MSC API will help mitigate this by implementing a double-buffering feature, essentially multitasking the USB and media sides of the transfer.)

15.18.2 Implementation Comments

USBMSC_config sets up this device as having removable media, since the SD-Card can be removed. This device has only a single-LUN, which is the SD-Card interface. As with all applications using this version of the MSC API, it only supports a Peripheral Device Type (PDT) of 0x00, which indicates "direct-access block device". This includes all devices based on magnetic or flash-based media.

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.

15.19 Example #M3: Multiple LUNs

This example demonstrates the implements 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.

15.19.1 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.

15.19.2 Implementation Comments

The number of LUNs is set by the Descriptor Tool. Once this is done, the application must configure *USBMSC_config* in such a way that it includes information for the number of LUNs selected within the Tool. The information for each LUN is the same as it has been for other MSC examples.

Both LUNs are set up as 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.

As with all applications using this version of the MSC API, it only supports a Peripheral Device Type (PDT) of 0x00, which indicates “direct-access block device”. This includes all devices based on magnetic or flash-based media.

Unlike the other examples, 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.

15.20 Example #CH1: Composite CDC+HID Device; Communicate Between Terminal and HID Demo App

15.20.1 Running It

This application shows a composite device consisting of a CDC and HID interface. To run it, an application called the *MSP430 HID Demo App.exe* is provided. Its function is similar to a terminal application, but it interfaces via HID instead of a COM port. In this example, 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 echos data between the interfaces.

Run the example and enumerate the device. As with the first time running one of the previous examples, Windows will generate a “Found New Hardware” box, initiating the *device installation* process. It’s important that the correct INF file be used, because the PID for this example is different than previous PIDs. 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, run Hyperterminal (or another terminal application), and open the proper COM port, as with the previous examples. Then also run the program *MSP430 HID Demo App.exe*. Set the VID to 0x2047 and the PID to 0x0302, and press “Set VID/PID”. A value should appear in the “Serial Number” and “Interface Number” menus. Press the “Connect” button to initiate a connection with the HID interface.

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

The value that appears in the “Serial Number” menu originates in the MSP430 device when it’s produced. Each physical MSP430 manufactured contains a unique die ID value. If the Descriptor Tool is configured to enable serial number reporting, the API automatically pulls this die ID and uses it to generate a unique serial number for this device. This value appears in this menu.

If the “Serial Number” and “Interface Number” fields don’t populate after pressing “Set VID/PID”, the device may not have enumerated properly. Check the Device Manager for a device listed under “Human Interface Devices”. Open “Properties”, and look in the “Details” tab. A string that includes the aforementioned VID/PID values should be present.

If this particular VID/PID pair has ever been associated with any other interface configuration on this machine (for example, CDC only), Windows may be confused about what this device is. Try changing the PID, by going into *descriptors.h* and changing the PID value to something new. Re-compile the code, and re-enumerate the device. Then change the PID value in the HID Demo App to match.

15.20.2 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.)

15.21 Example #CC1: Composite CDC+ CDC Device; Communicate Between Two Terminal Apps

15.21.1 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.

Run the example and enumerate the device. Unlike previous example, there should be *two* device installation procedures, since there are two CDC interfaces. 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. This INF contains the necessary information for having two CDC interfaces. (See Sec. 15.2.4 for more information about how Windows handles INF files.)

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.

15.21.2 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.

15.22 Example #HH1: Composite HID+HID Device; Communicate Between Two HID Demo App Instances

15.22.1 Running It

This application is just like the preceding ones, except it shows a composite HID+HID device. Run the example and enumerate the device. Since both interfaces are HID, there will be no "Found New Hardware" boxes – both interfaces will load silently.

Run two instances of the *MSP430 HID Demo App.exe* application. Set the VID in each one to 0x2047, and the PID to 0x0314. Press "Set VID/PID" in each instance's window. After a few seconds, a value should appear in the "Serial Number" menu, and "HID0" should appear in the "Interface" menu, of both windows. In one of the windows, select interface "HID1" instead. 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.

15.22.2 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.

15.23 Example #CHM1: Composite CDC+HID+MSC Device; Communicate Between Communicate Between Terminal and HID Demo App, with Two Storage Volumes

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.

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

16 For More Information

MSP430 landing pages:

- MSP430 USB landing page (http://www.ti.com/msp430usb)
  - Check here for all MSP430 USB software, tools, and datasheets
- MSP430 landing page (http://www.ti.com/msp430)
  - Check here for general MSP430 documentation and software
- Product pages for individual USB-equipped MSP430 devices
  - Check here for information specific to a given MSP430 device derivative
  - From any page on http://www.ti.com, enter the device number in the 'device search' field)

Documentation for MSP430 hardware architecture, including the USB module:

- MSP430F5xx User's Guide (http://www.ti.com/msp430)

Datasheets for specific USB-equipped MSP430 devices, including device-specific information and parametric data:

- MSP430F5529 datasheet (http://www.ti.com/lit/gpn/msp430f5529)
- MSP430F550x datasheet (http://www.ti.com/lit/gpn/msp430f5510)
- MSP430F563x datasheet (http://www.ti.com/lit/gpn/msp430f5638)
- MSP430F663x datasheet (http://www.ti.com/lit/gpn/msp430f6638)

USB specifications:

- USB 3.0 specification (http://www.usb.org/developers/docs/)
- Communications Device Class specification and PSTN Subclass specification (http://www.usb.org/developers/devclass_docs#approved)
- Device Class Definition for Human Interface Devices (v1.11) (http://www.usb.org/developers/devclass_docs#approved)

17 References

- *Starting a USB Design with MSP430 MCUs (http://www.ti.com/msp430usb)*
- *MSP430 Software Coding Techniques (slaa294)*
- USB 3.0 specification (http://www.usb.org/developers/docs/)
- Communications Device Class specification and PSTN Subclass specification (http://www.usb.org/developers/devclass_docs#approved)
- Device Class Definition for Human Interface Devices (v1.11) (http://www.usb.org/developers/devclass_docs#approved)

Appendix A. Configuration Constants

Various aspects of the API are controlled with *configuration constants* defined in *descriptors.h.* They are intended to be controlled with the Descriptor Tool, but they are also documented here. They are compile-time values that do not change during operation.

Table 64. USB-Specific Configuration Constants

Constant	Description
USB_VID	Four-digit hex value reflecting the USB vendor ID, as assigned by the USB-IF.
USB_PID	Four-digit hex value reflecting this device's product ID, as assigned by the OEM.
USB_STR_INDEX_SERNUM	Indicates the index of the string descriptor that contains the serial number string. For information regarding the tradeoffs of reporting a serial number, see the Descriptor Tool's help pane. If this value is non-zero, then a string descriptor must be located at that index. If no serial number is desired, this index should be assigned a value of 0.
USB_SUPPORT_REM_WAKE	Controls whether the remote wakeup feature is supported by this device. A value of 0x20 indicates that it is supported. A value of zero indicates remote wakeup is not supported. Other values are undefined and could result in unpredictable behavior.
USB_SUPPORT_SELF_POWERED	If the device is self-powered to any degree, this constant should be set to 0x40 (default value). If it is fully bus-powered while attached to the host, it should be set to 0x00.
USB_MAX_POWER	This is the value to be reported in the *bMaxPower* field of the configuration descriptor. The maximum amount of current the device will draw from the host, in mA, is the value of *USB_MAX_POWER* times two. (a value of 50 = 100mA).
USB_NUM_CONFIGURATIONS	The current version of the API restricts this value to 1.
USB_DMA_CHAN	Selects the DMA channel to be used for USB operations. Must reflect a channel that is implemented in the chosen MSP430 derivative. 0x00 for ch. 0, 0x01 for ch. 1, up to 0x07 for ch. 7. If no DMA channel is to be used, rather relying on CPU transfers, set this to 0xFF.
USB_PLL_XT	This selects the high-speed oscillator used to source the PLL. It must have a value of either 1 (for XT1) or 2 (for XT2). All current MSP430 USB devices use XT2.
USB_DISABLE_XT_SUSPEND	If non-zero, then the API will disable the oscillator indicated by USB_PLL_XT in response to a USB suspend. If zero, then a USB suspend will not affect the oscillator. *USB_resume()* will always restore the oscillator in either case.
USB_MCLK_FREQ	The MCLK frequency while USB is operating, in Hz. (12500000 = 12.5MHz) The API uses this only to determine delays during startup of the USB module; it does not configure MCLK. This must be done outside the application. If MCLK isn't always the same speed during operation, this constant should reflect the highest value ever in use.
VER_FW_H / VER_FW_L	The high and low bytes of the device release number, which is reported in the *bcdDevice* field of the device descriptor. As the name implies, the values are binary-coded decimal.

Table 65. Interface-Specific Configuration Constants

Constant	Description
MAX_PACKET_SIZE	The size of packets used to exchange serial data over USB for CDC. Larger packets are generally more efficient when transceiving large chunks of data. Must be set to 64.
USBHID_POLL_INTERVAL	Interval at which the host will poll this device for HID reports. Can be 1 to 255. By default, it is 1, which maximizes throughput.
MSC_MAX_LUN_NUMBER	The number of LUNs in an MSC interface, minus one. (0 means there is one LUN, 1 means two LUNs, etc.)

Appendix B. Supported SCSI Commands

The MSC API automatically handles all SCSI commands required to function with the target host operating systems, based on information it receives from the application. The list of SCSI commands includes:

- INQUIRY
- REQUEST SENSE
- TEST UNIT READY
- READ
- WRITE
- READ CAPACITY
- MODE SENSE
- REPORT LUNS