Programmer's Guide: MSP430 USB API & Descriptor Tool

MSP430

ABSTRACT

The MSP430 USB API implements four USB device classes: the Communications Device Class (CDC), the Human Interface Device (HID) class, the Mass Storage class (MSC), and the Personal Healthcare Device Class (PHDC). It's designed for easy creation of USB applications on MSP430 microcontrollers that are equipped with an on-chip USB module.

Contents

1 Introduction

1.1 Overview

The USB API (application programming interface) stack for the MSP430 is a turnkey API. It's intended to enable easy and reliable creation of a simple USB data connection between an MSP430 and a USB host. It includes support for these USB device classes:

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

The API is designed to minimize the USB knowledge required to write an application:

- All USB protocol is handled automatically by the API
- The data interface presented to the application is simple to use, abstracting the application from USB protocol
- 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 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 in the MSP430 USB Developers Package.

1.2 The MSP430 USB Developers' Package

This Programmer's Guide documents the USB API, Descriptor Tool, and USB examples. However, these are part of a larger collection: the USB Developers Package.

The table shows the contents of the Dev Package. The organization in the table matches the directory structure of the Dev Package files; but the names are expanded, for human readability.

Table 1. MSP430 USB Developers Package Contents

Item	Description
Host USB Software	
Java HID Demo App	Host-side example for implementing HID. It complements the MSP430's HID-Datapipe API, simplifying creation of a general-purpose USB HID device.
Python USB Firmware Upgrader	One of two host-side USB firmware update solutions provided TI-MSP430. (The other is based on Microsoft Visual Studio and not included in the USB Dev Package; see the note below this table.) This upgrader is not documented within this Programmer's Guide. See the upgrader's directory in the USB Developers Package for documentation.
MSP430 USB Software	
Documentation	
API Functional Reference	Doxygen-generated function reference for all the USB API calls
Examples Guide	A guide for the person *evaluating* the USB API, using the examples
MSP430 USB API Programmer's Guide	This document. A detailed reference for someone that's decided to *develop* a USB device using the USB Developer's Package.
Release Notes HTML file	
MSP430 USB API	API software library for implementing USB devices
Examples	Contains anything unique to a particular revision, including bug fixes, benchmarks, compiler dependencies, and migration information.
emptyUsbProject	A project containing only a main.c populated with a suggested main loop framework. The framework is commented with instructions.
CDC Examples	Examples for implementing virtual COM ports using the CDC class.
HID Examples	Examples using the HID class
HID-Datapipe	Examples of HID-Datapipe, a means of UART-style data exchange based on the HID class.
HID-Traditional	Examples of ordinary HID interfaces, like mice and keyboards
MSC Examples	Examples for implementing mass storage devices, like SD-card and emulated on-chip flash drives.
Composite Examples	Examples of *composite USB devices;* that is, devices comprising more than one of the interfaces above
SYS/BIOS Examples	Examples showing us of the USB API with TI's SYS/BIOS RTOS
driverlib (driver library)	Contains the standard MSP430 driverlib. It is referenced by all the examples.
USB_API	These are the actual USB API code files. It is referenced by all the examples.
MSP430 USB Descriptor Tool	Automatically generates reliable descriptors for literally any combination of USB interfaces. It saves the developer's time, and reduces the chance for errors. Using the Tool is part of the standard recommended design flow.
USB Dev Package Manifest	Information related to licensing and origins of the code.

Separate from the USB Developers Package is an application note *MSP430 USB Field Firmware Updater (slaa452)*, and its associated firmware update host application based on Microsoft Visual Studio. This application note discusses design considerations when implementing updates over USB using the MSP430's on-chip bootstrap loader (BSL). It especially targets applications where the end user performs this in the field. This updater based on Visual Studio is completely unrelated to the Python-based one above.

2 Introduction to the MSP430 USB API

2.1 MSP430 USB API Stacks: Overview

The MSP430 API stack allows easy creation of USB devices on MSP430 derivatives equipped with an on-chip USB module. The architecture emphasizes compact code space and light execution. Resource usage outside the USB module is minimized, allowing easy integration into the end application.

2.1.1 Devices Classes Supported by the MSP430 API Stacks

A USB device must contain one or more USB interfaces. Each interface must be of a particular *USB* device class type. Device classes define a USB protocol to support a given type of device.

Table 2. Supported USB Device Classes

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

CDC, HID, and MSC are often used in general-purpose USB applications. In contrast, PHDC is very specific to Continua applications, and requires higher-level software layers to implement Continua compatibility.

All of these share a common USB layer.

2.1.2 Development Environments Supported by the API

The USB API stack and examples build and run on both the IAR and CCS environments for MSP430. See the Release Notes HTML file in the USB Developers Package for specific IAR/CCS version information.

Support for GCC is available only on CCS 6.0 and v4.10 of the USB API stack supports MSP430 GCC natively.

IAR and CCS are both available in free, code-size-limited versions (8K and 16K, respectively, of object code). Applications that fit under 8K of memory can be run on both free versions. Applications that are greater than 8K cannot be built using the free IAR Kickstart tool. Instead, the free version of CCS can be used; or a licensed version of either environment.

See the Release Notes within the USB Developers Package zip file for additional information specific to a given release.

2.1.3 Hardware Requirements

The API stack and examples run, unmodified, on all four USB-equipped MSP430 families:

- F552x/551x
- F550x (including the F5510)
- F663x/563x
- F665x/565x

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

The only hardware configuration required is to select the appropriate MSP430 derivative in the project settings. A few of the examples use buttons or an SD-card, which may be hardware-dependent. See the Examples Guide within the USB Developers Package for step-by-step instructions for configuring this. For starting a new USB project, see Sec. 4.

2.1.4 Host Operating System Support

All the USB device classes supported by the USB API are supported natively within Windows, MacOS, and Linux distributions. This has been true for over ten years. For specific information about the versions against which the API has been tested, see the Release Notes HTML file within the USB Developers Package.

Native host OS support has several 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

For CDC, there are considerations unique to Windows and the Mac OS. Sec. 2.2.2 discusses this in detail.

Since virtual COM ports (CDC) and storage volumes (MSC) are generic interfaces supported by all these host operating systems, there's a wealth of support in the public domain for how to write host applications interfacing to them. In contrast, HID is more specific to USB, and less familiar to most developers. To help with this, the MSP430 USB Developers Package includes a Java-based HID application called the *Java HID Demo App*.

2.1.5 USB Certification

The MSP430 device derivatives, running the USB API, have passed USB certification testing for all four device classes (CDC/HID/MSC/PHDC). All certification was performed at MCCI.

An output of the USB certification process is a *test ID,* or *TID.* The TID's for all current MSP430 silicon is shown in the table below.

Table 3. MSP430 Device Test IDs (TIDs)

MSP430 Derivative	Package	TID
MSP430F552x	PT/RGC/RGZ (QFP/QFN)	40000973
	ZQE (BGA)	40001139
MSP430F550x/5510	PT, RGC, RGZ (QFP/QFN) (Note: F5504RGZ not included in this TID; see below)	40001138
MSP430F5504RGZ	RGZ (QFN)	41001138
MSP430F563x	PZ (QFP)	40001250
MSP430F663x	ZQW (BGA)	40001442
MSP430F565x	PZ (QFP/QFN)	40001444
MSP430F665x	ZQW (BGA)	40001443

All MSP430 devices were certified under MSP430's own vendor ID, 0x2047. This VID is separate from TI's main VID (0x0451).

2.1.6 driverlib Support

Starting with v4.0 of the USB API, projects now depend on the MSP430 driverlib ("driver library") HAL layer. driverlib brings portability and standardization among MSP430's software offering, especially for basic MCU functions like clocks and power.

For those porting applications from earlier versions of the API, the move to driverlib will not be very noticeable. The application examples in the USB Dev Package, however, have all been updated to include driverlib. The "HAL core library" used by previous versions has been deprecated by TI MSP430. Please see the Release Notes HTML file for migration information.

Besides being included in the USB Developers Package, driverlib can be obtained for general MSP430 development with MSP430Ware.

2.1.7 Stack Organization

The software stack is shown in the figure below.

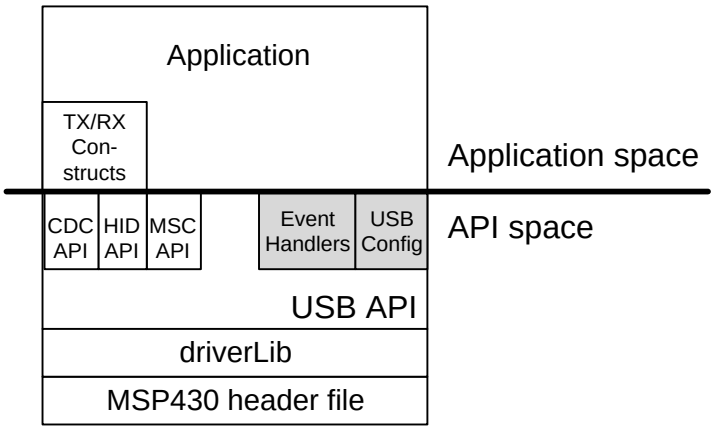


Figure 1. MSP430 API Stack Diagram

The individual device class layers (CDC, HID, MSC) all share a common USB layer (USB API). The application makes calls to these APIs, and also directly to the USB layer.

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

The sections “USB Config” and “Event Handlers” are special spaces that straddle API/application space (and thus are shaded grey). They are core to the API, but are controlled by the user.

The files are shown in the table below. The table’s organization reflects the actual file/directory organization.

Table 4. MSP430 USB API Stack Files

File/Directory			Description
Standard MSP430 libraries	driverlib		Used to configure power and clocks, and to read the device's TLV structure (Tag-Length-Value) in MSP430 flash, to obtain the device's unique die ID number.
	MSP430 header file (for example, *msp430f5529.h*)		Standard header file for the particular MSP430 device derivative being used. Ships with IAR/CCS. Selection is automatically controlled by project settings.
USB API Space	USB_API		
	USB_Common		
		defMSP430USB.h	Definitions related to the MSP430 USB module
		device.h	Helps configure the stack for the device derivative selected in the IAR/CCS project settings.
		types.h	Deprecated. The USB stack has been converted to use C99 types.
		usbdma.c	Functions related to DMA transfers
		usb.c/h	Functionality common to all USB applications.
		UsbIsr.h	Header file for *UsbIsr.c,* located in \USB_Config
	USB_CDC_API		CDC-related functionality
		UsbCdc.c/h	
	USB_HID_API		HID-related functionality
		UsbHid.c/h	
		UsbHidReq.c/h	
	USB_MSC_API		MSC-related functionality
		UsbMsc.h	
		UsbMscReq.c/h	
		UsbMscScsi.c/h	
		UscMscStateMachine.c/h	
	USC_PHDC_API		PHDC-related functionality
		UsbPHDC.c/h	
USB Config Space	USB_config		Configures the API's characteristics, including what interfaces the device contains. Generate these files using the *MSP430 USB Descriptor Tool*.
	descriptors.c/h		Contains the device's USB descriptors and other information that configures the API.
	UsbIsr.c		USB interrupt service routine handler, and related functionality
Application Space	USB_app		
	usbEventHandling.c		Event handlers, for responding to USB-generated events. Write application-specific code into these handlers.
	usbConstructs.c/h		Example constructs for send/receive operations. The functions here reflect the approaches described in Sec. 11
	(other application files)		Other user-defined application files.
	hal.c/h		Included with the USB examples. It contains hardware-specific adjustments for the F5529 Launchpad, F5529 Experimenter's Board, and all relevent FET target boards, making it simple to run the examples on any of these boards.
	main.c		User application

2.1.8 Usage of MCU Peripheral Resources

Within the USB API, the resources shown below are considered to be owned by the API. If the application accesses them, it should be aware of how the API uses them.

Table 5. Resource Ownership

Resource	Owned by the API When	How It's Used
USB module	Always	Generally speaking, the application should not attempt to control the USB module's registers or pins (see note)
XT2 oscillator	Always	The API configures, starts, and stops XT2 as it needs, in order to perform USB communication. The behavior can be modified by the `USB_DISABLE_XT_SUSPEND` and `USB_XT_FREQ` configuration constants (handled by the Descriptor Tool). Although the application can use XT2 to drive the CPU's MCLK, there's usually little value to this, and it's not recommended.
DMA channel	While both conditions are true: • DMA functionality is enabled using `USB_DMA_CHAN` (via Descriptor Tool). • the USB state is `ST_ENUM_ACTIVE`	During these times, the API owns this channel, used for both TX/RX. The application should not attempt to use it.

Note: 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.

These are the only peripherals used by the API. The only other resources used are memory, and CPU cycles.

2.1.9 Memory Requirements

The MSP430 USB API object code is compact and uses relatively few cycles to execute. Please see the Release Notes accompanying the source code, for code size information specific to a particular code release.

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

The API's RAM usage is static.

2.1.10 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. In the accompanying USB application examples, examples are provided showing use of the API with TI's SYS/BIOS RTOS.

2.1.11 Support for Composite USB Devices

The MSP430 USB API is unusually easy to configure for any combination of USB interfaces (i.e., in creating a composite USB device). All that's required is to use the Descriptor Tool to select the interfaces; the API becomes automatically configured. The resulting USB descriptors are in the exact format required by Windows, the MacOS, and Linux. All that remains is to write the application. (See Sec. 3 for more information about the Descriptor Tool.)

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, 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 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 the vast majority of applications.

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

2.1.12 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.

Another concern that can affect bandwidth on any USB device is software on the USB MCU. The MCU application must prepare data to be sent, and process data that's arrived; this can put the CPU in the critical path, slowing bandwidth.

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. Running at the maximum MCLK can often achieve bandwidth close to what is shown for DMA.

2.1.12.1 Considerations for CDC/MSC Interfaces

If bulk transfers are used – for example, as CDC and MSC interfaces do -- then the speed is further impacted by host/bus conditions:

- 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. Bulk transfers 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.

Because high bandwidth are often desirable for CDC/MSC interfaces, TI does benchmarking of these interfaces. The results are shown in the Release Notes.

2.1.12.2 Considerations for HID Interfaces

Interrupt transfers – used by HID interfaces -- are immune to these effects; they maintain steady bandwidth, no matter what else is happening on the bus. However, they're also limited in bandwidth, to 64KB/sec. As a result, if the required bandwidth is low and the bus may experience heavy loading, HID's steady bandwidth can be an advantage.

For an HID interface to achieve this 64KB/sec, the polling interval must be set to 1ms. (Each USB packet is 64 bytes.) This can be done with the Descriptor Tool (see Sec. 3). The polling interval determines the rate at which the host will exchange data with this interface.

If 64KB/sec isn't fast enough, the device might be able to still use HID by creating multiple HID interfaces in composite. The Descriptor Tool can easily do this. The MSP430 and host applications could then both be written to interleave their data across them.

2.1.13 Release Notes, and Migration from Previous Versions

A "Release Notes" HTML file accompanies each release of the USB Developers Package. Reference this file for any information specific to this release, including:

- All changes from the previous versions
- Instructions for migration from previous versions
- Code size / memory calculations
- CDC/MSC performance benchmarks
- Updated IDE configuration information
- Known issues

2.2 The Communications Device Class (CDC) API

2.2.1 CDC Overview

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

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 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.

2.2.2 Host Considerations

CDC support varies somewhat among the major host operating systems, which is described below.

Placing a CDC interface alongside other interfaces in a composite USB device is a little problematic. The CDC specification didn't leave room for this, which has forced the need for other solutions. The official, supported solution is the use of an auxiliary USB descriptor called the *Interface Association Descriptor* (IAD). This has been adopted by Windows and Linux; but it has not been adopted by the MacOS.

2.2.2.1 Microsoft Windows

The CDC class is supported by all versions of Windows going back to Windows 2000.

Although Windows natively contains the driver binaries, it doesn't contain an INF file. (Windows uses INF files as a means of associating devices with drivers.) 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 must somehow be guided to the INF file. The Descriptor Tool generates an appropriate INF file when generating its output, so there is no need to create one manually.

If the device is composite, then the INF file needs to be tailored to the chosen set of USB interfaces. The Descriptor Tool accounts for this when it generates its INF file.

In addition to a CDC driver, the host needs an application to interface with the resulting virtual COM port. Although a custom application is probably desirable, any "terminal" application can be used to communicate with an MSP430 equipped with the CDC interface.

Windows XP Service Pack 3, Vista, and 7 all support the IAD, and thus they're able to support CDC interfaces alongside other interfaces within a composite USB device.

However, earlier versions of Windows do not support the IAD, and therefore they do not support composite CDC devices. The Descriptor Tool will alert the user if this is attempted.

2.2.2.2 MacOS

The CDC class is supported on any recent version of the MacOS.

However, the MacOS does not support the IAD. As a result, it is not possible for a Mac to enumerate a composite device that contains a CDC interface; only single-interface CDC devices are supported. The Descriptor Tool will alert the user if this is attempted.

2.2.2.3 Linux

The CDC class is supported on any recent, common Linux distribution.

Recent versions of Linux support the IAD, and therefore can work with composite CDC devices.

2.3 The Human Interface Device (HID) API

2.3.1 HID Overview

The Human Interface Device class is perhaps the oldest and most established USB device class, in that it was created for use with the most basic USB application: 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 several advantages, discussed below.

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.

When using HID for general-purpose applications, it's soon discovered that developing host support for HID isn't like CDC/MSC. CDC/MSC generate interfaces on the host that are extremely common and popular (virtual COM port or storage volume, respectively). In contrast, host operating systems don't generate a similar generic interface for HID. For many host programmers, then, HID represents a small learning curve. To help with this, TI makes available the *Java HID Demo App*, which is designed to pair with the MSP430 HID API stack.

There is a sub-protocol in HID called the boot protocol. This is a "light" version of USB HID that can be run from a PC's BIOS program, prior to loading the OSes' full HID drivers. This allows mice and keyboards to begin operating from the very beginning of the boot process. This protocol is supported by the API.

Some legacy HID implementations request the host to send data via *control endpoint zero.* Usually, the host only uses this endpoint for management functions. Note that endpoint zero on the MSP430 is only eight bytes wide (compared to 64 for the others). API functions are provided to enable compatibility with these legacy implementations.

2.3.2 HID-Datapipe vs. HID-Traditional

At a basic level, all HID interfaces send/receive *HID reports.* Unlike freeform data packets, reports are heavily formatted, using a complex scheme not unlike a scripting language. This script is placed within a *report descriptor,* reported to the host during enumeration.

There is a learning curve associated with this. Report organization is not flat; it is filled with concepts such as *collections* and *usages,* which can be placed inside each other to several levels.

Since there are otherwise advantages to using HID for general-purpose use, TI creates a special predefined HID configuration called *HID-Datapipe,* which use a special set of function calls. To the MSP430 application, this interface appears as a UART-style unformatted datastream. And in fact, these calls are identical to the ones used with CDC interfaces. The application prepares a data buffer of any size, and sends/receives the entire chunk. Underneath the HID-Datapipe calls, a special report format is used that is comprised only of two fields: a *size* byte, with the remainder of the USB packet reserved for *data*. In other words, this report uses the simplest possible configuration, allowing the application to format the data however it wishes.

Any HID interface that isn't a datapipe is called *traditional.* Traditional HID interfaces send/receive reports. The developer must provide a report descriptor, and use the HID-Traditional function calls to send/receive reports. Some engineers need to create traditional HID interfaces in order to use previously-existing host HID applications, or because the host operating system interacts with the device directly (like mice or keyboards).

The developer enters the report descriptor into the Descriptor Tool. The Tool contains pre-defined report descriptors for mice and keyboards. Any other traditional HID interface is termed 'custom', and the developer must enter the raw report descriptor.

Sec. 7 describes datapipe interfaces, including CDC and HID-Datapipe. Sec. 9 describes traditional HID interfaces.

2.3.3 Host Considerations

Unless the device being created is one the host operating system interacts with (like a mouse or keyboard), a host application will be required. Unlike CDC, which can be used with ordinary "terminal" applications, a HID interface generally requires a custom application (unless it's of a type recognized by the host OS).

TI provides the *Java HID Demo App* to assist with this. By default, it supports the HID-Datapipe model and HID report format. However, it could be adapted for customized reports as well. Being Java, it runs on Windows, Mac, and Linux, provided a Java Runtime Engine is present.

2.3.3.1 Microsoft Windows

The HID class is supported by all versions of Windows going back to Windows 2000.

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 trouble free for the end user.

2.3.3.2 MacOS/Linux

The HID class is supported on any version of the MacOS or Linux. See the Release Notes HTML file for specific version information.

2.4 The Mass Storage Class (MSC) API

2.4.1 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. 2.4.4 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.

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

Compared to CDC/HID, the application carries a heavier role: it must implement the volume to be mounted. The API can’t handle this itself, because there are many types of media that might get implemented. If the application doesn’t do this, the MSC interface might enumerate, but the volume won’t mount in the system.

Unlike CDC/HID, only one MSC interface can be created within a physical USB 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.

2.4.2 Storage Device Types

The API implements two sets of SCSI commands, selectable by the application. Each is associated with a particular *peripheral device type* value; the host treats these types differently.

Table 6. Storage Device Types

Device Type	SCSI Command Set	Description
Direct-Access Block Devices	SCSI Block Commands (SBC)	Hard drives, flash drives, memory cards
CD-ROM/DVD	MultiMedia Commands (MMC). (Not to be confused with MultiMedia Cards, which use the SBC command set.)	CD-ROM and DVD players.

The developer selects this in the Descriptor Tool. The tool will automatically use the correct SCSI command set. Since the host uses a different file system for each, the implementation of the volume may need to be different.

2.4.3 File System Software

If the MSP430 application needs to comprehend files on the medium -- rather than simply handling the host's access requests -- file system software must be added. (This is discussed in more detail in Sec. 8). The API is designed to be usable with any file system the developer chooses, and many third party and open source options are available. But as an example, the USB Developers Package's MSC examples demonstrate usage with the popular open-source FatFs file system software

2.4.4 Host Considerations

To access the volume on any host operating system, a host application is required. Writing such an application is generally straightforward, because of the wealth of information and tools available for file access and storage volumes. (In this, the situation is similar to that of virtual COM ports.)

2.4.4.1 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 a user-provided INF file.

2.4.4.2 MacOS/Linux

The MSC class is supported on any modern version of the MacOS or Linux. See the Release Notes HTML file for specific version information.

Like on Windows, MSC interfaces load silently on Mac/Linux.

2.5 The Personal Healthcare Device Class (PHDC) API

PHDC is the USB layer within a USB device that supports the Continua Health Alliance. In such a device, the data follows the data/messaging standards ISO/IEEE 11073-20601 + ISO/IEEE 11073-10407. Note that these IEEE layers are not provided with this software package. Texas Instruments offers a complete Continua-certified PHDC solution.

Note that it isn't supported to have more than one PHDC interface in a composite USB device; the Descriptor Tool will prevent this from being implemented.

Because of the tight association of PHDC with Continua, it is not treated in the USB Developers Package as CDC/HID/MSC are. High-level information about PHDC is provided in this Programmer's Guide, as well as how to create PHDC interfaces. (PHDC is fully supported by the Descriptor Tool.) However, this Programmer's Guide doesn't describe how to interact with a PHDC interface, and the USB examples don't include any for PHDC. For these resources, see MSP430's Continua page.

3 Getting Started With Evaluation

To get started with evaluating and studying the MSP430 USB API, refer to the Examples Guide, also within the USB Developers Package. It contains all information needed to evaluate the examples.

The example set includes:

- 7 CDC (virtual COM port) examples
- 7 HID-Datapipe examples
- 2 traditional HID devices (a mouse and a keyboard)
- 4 MSC examples
- 4 composite examples
- 2 examples using TI's SYS/BIOS RTOS
- 1 example for how to reduce interrupt latency

The Examples Guide contains information on:

- Step-by-step directions for loading and running the examples, on CCS or IAR
- The dynamics of INF files on Windows for CDC interfaces
- Using the Java HID Demo App

In short, it contains the information needed by the person only *evaluating* the USB Developers Package, while this Programmer's Guide contains information for the person beginning *development*.

4 Getting Started With Development

Three separate approaches to beginning development with the USB API are described here:

- Starting development from an existing example
- Starting development from an “empty” USB API project
- Adding the USB API into an existing MSP430 application

The first two approaches are advantageous for new development. When an application already exists, into which USB is being added, the third approach may be necessary.

It’s assumed that the reader is using a recent version of IAR Embedded Workbench, or Code Composer Studio v5.5 or later.

The details of CCS/IAR interaction are not discussed here; please refer to the appropriate documentation, as needed.

4.1 Starting Development from an Empty USB Project in the TI Resource Explorer

The TI Resource Explorer contains a wizard that helps you create a new USB API project.

The wizard performs these steps:

1) Creates an empty USB project
2) Walks you through using the Descriptor Tool to create your USB interfaces
3) Prompts you to adjust the MSP430 device derivative, for your hardware
4) You can begin developing your application!

The wizard is shown below.

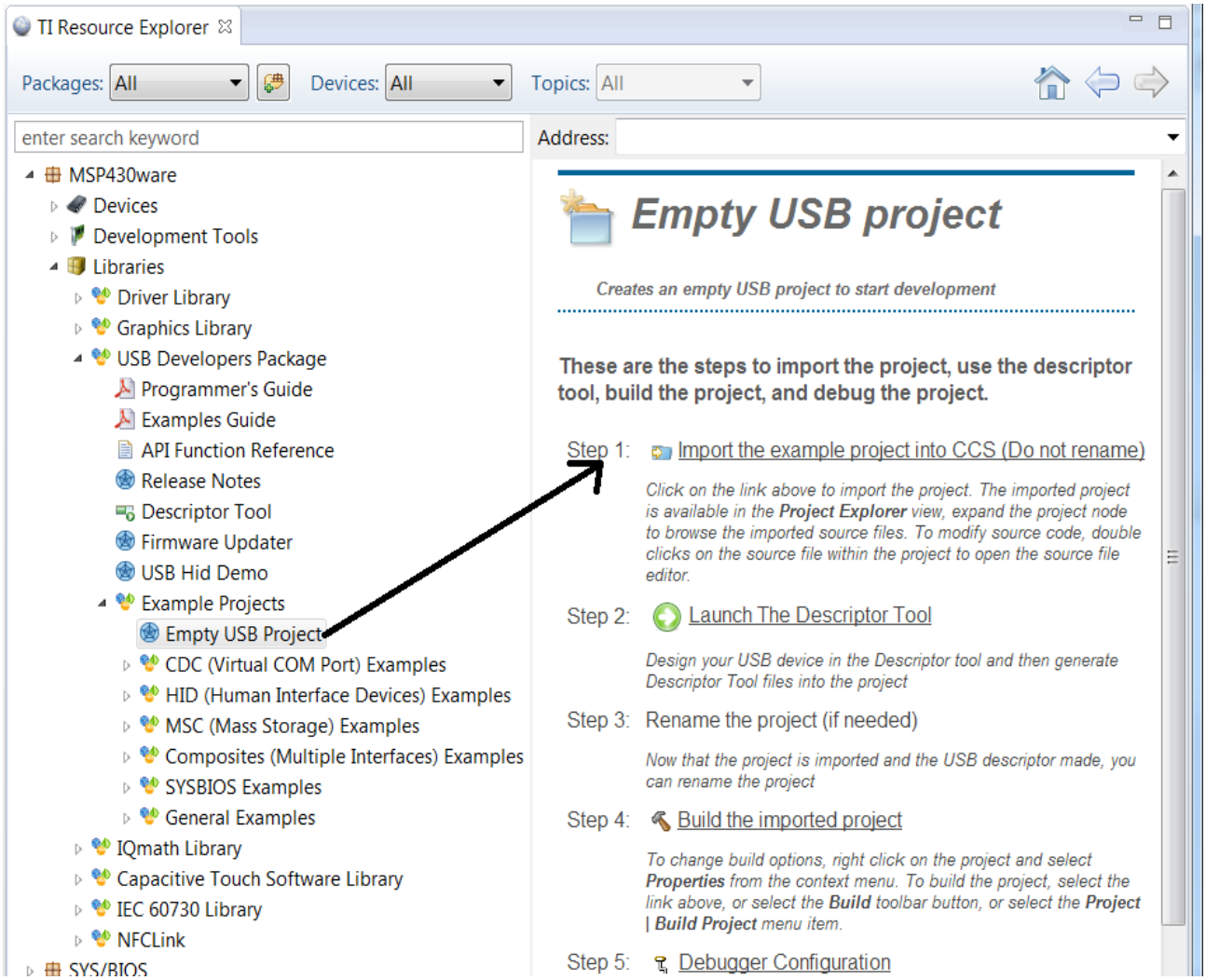


Figure 2. TI Resource Explorer: "Create an Empty USB Project" Wizard

After the wizard completes, it provides a framework main(), including the suggested main loop framework discussed in Sec. 11.8. You can insert code directly into this framework, borrowing from examples and referencing this Programmer's Guide.

In this empty project, the recommended project settings and source directories are already in place, with no need to edit them.

4.2 Starting Development from an Empty USB API Project

IAR users not having access to the wizard discussed above may wish to copy or import the project emptyUsbProject in the \examples directory of the USB Developers Package, and begin development from this project. This is the same project as is produced by the wizard, except that the Descriptor Tool has not been run. You can accomplish the same effect by running the Descriptor Tool by double-clocking on its executable file, and depositing the generated output in your target project.

As with starting from an example, the recommended project settings and source directories are already in place, with no need to edit them.

It's recommended to make a copy of this project, rather than using it directly, in case the empty project is desired again at a later time.

4.3 Starting Development from An Existing Example

It's also possible to start development from an existing example project; by removing non-applicable example code, and adding new code.

If you find an example that has your desired USB interfaces already defined, then for the time being, you may not need to run the Descriptor Tool. Although this is enough for experimentation or simple development, bear in mind that if taking a product to market, additional tuning of the descriptors will eventually be needed. The Tool is a good way to do that.

Be aware that not all the necessary code is located in the specific example's directory; the \driverlib and \USB API directories, external to the example's directory, are required.

If using IAR and wanting to make a copy of the example, manually copy the directories to a new location: the example's directory, \driverlib directory, and \USB_API. Be sure to keep the same directory structure that the example used.

If using CCS, simply import the example into your workspace. Beginning with v4.0 of the USB Developers Package, the examples are based on the "projectspec" feature present in CCS v5.5 and later. The .projectspec file present in each example's directory will cause a CCS import to always result in a new copy of the example. It will replicate the same directory structure used by the example.

4.4 Adding USB into an Existing MSP430 Application

When USB is being added into an existing application, the appropriate USB API directories must be manually copied into the target project, and merged.

Various directories within the USB API projects are discussed below; please reference Table 4.

4.4.1 Adding in the Files

The USB API core files, located in the \USB_API directory, must be added. It's easiest to simply add the entire directory \USB_API, as a whole, to the target project. Files within this directory have relative #include pathways, and won't be broken by this copy.

1) The files *descriptors.c/h* and *UsbIsr.c* are typically located in a directory \USB_config, outside of \USB_API. This is because of the special role they play in configuring the API, and because their files originate from outside the API – from the Descriptor Tool. If desired, the developer can put these files anywhere. To support this, their path has not been hardcoded into the API; the #include path for these files is kept in the project settings. If moving them to a different directory, the developer must adjust this path.

2) It's suggested that applications using CDC and HID-Datapipe interfaces use the construct functions in *usbConstructs.c/h*, to send and receive data. Copy these into the target application.

3) Copy \driverlib into the target project. As of v4.0 of the USB API, it's now based on driverlib. driverlib may also be useful to the application, for accessing peripherals. Note that the version of driverlib distributed in the USB Developers Package is only a subset of the full distribution; MSP430Ware contains the full version of driverlib.

Although the USB examples use a \USB_app directory, there's no requirement for how the application places its own files.

The files are now present in the target directory. #includes must be added to ensure the files are found.

4.4.2 Managing #includes

The application will need to #include several files for basic USB functionality:

```
#include "descriptors.h"
#include "USB_API/USB_Common/device.h"
#include "USB_API/USB_Common/usb.h"
```

4) #includes to *.h files for the selected USB interfaces must be made. The directories are \USB_CDC_API, \USB_HID_API, and/or \USB_MSC_API. For example, if your application has a CDC interface, be sure to include every *.h file in \USB_CDC_API, within any file referencing CDC API function calls. The same applies to HID and MSC.

5) Applications using the CDC/HID-Datapipe constructs in usbConstructs.c will need to include usbConstructs.h.

Having taken these steps, the project code should be ready for USB development.

As discussed above, multiple #include links are made within the \USB_API directory, from one file to another within \USB_API. It's recommended not to make organizational changes within this directory.

The files normally placed in \USB_config – *descriptors.c/h* and *UsbIsr.c* – can actually be located anywhere. But if moving them out of \USB_config, the include path for these files in the project settings must also be changed.

4.4.3 Project Settings

Add global paths to:

- the directory *containing* \USB_API. Don't include \USB_API itself, just the directory that contains it. (In the USB examples, this is the project root directory.)
- the directory containing the Descriptor Tool output files. (In the USB examples, this is \USB_config.)
- For driverlib, add an include for driverlib\MSP430F5xx_6xx

Stack size must also be managed; the USB examples can be used to get an idea of the requirements, which are generally modest.

5 MSP430 USB Descriptor Tool

The Descriptor Tool is a part of doing development with the MSP430 USB API. It 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.

The Tool has built-in help – both an overview sheet that's available from the Help Menu, and also contextual information in the Help Pane. But, additional information is provided here.

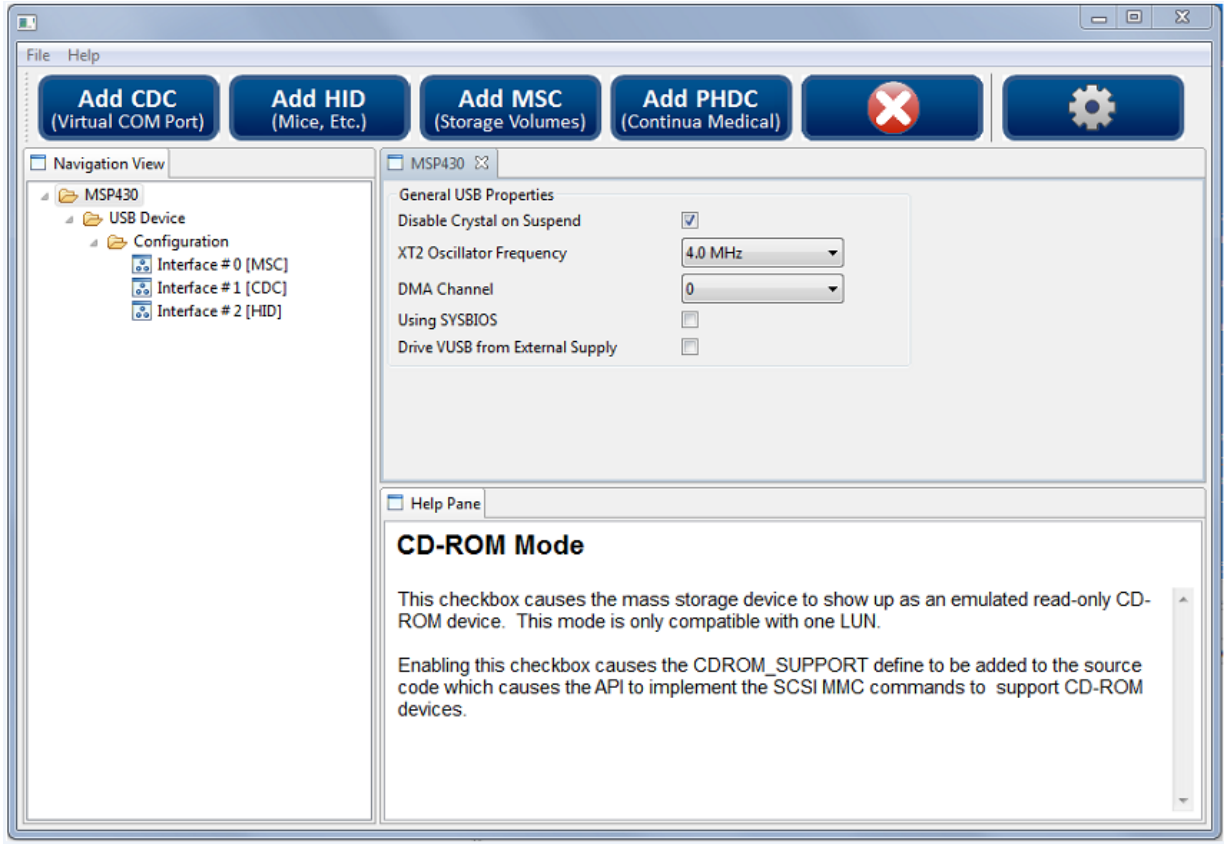


Figure 3. The Descriptor Tool

5.1 What is the Tool?

The Descriptor Tool generates a set of USB descriptors for your project. In so doing, it also builds the *USB interfaces* your application can use to interface with the host. After it runs, you can write your application to send/receive data over these interfaces.

An example of a USB device with three interfaces is shown below.

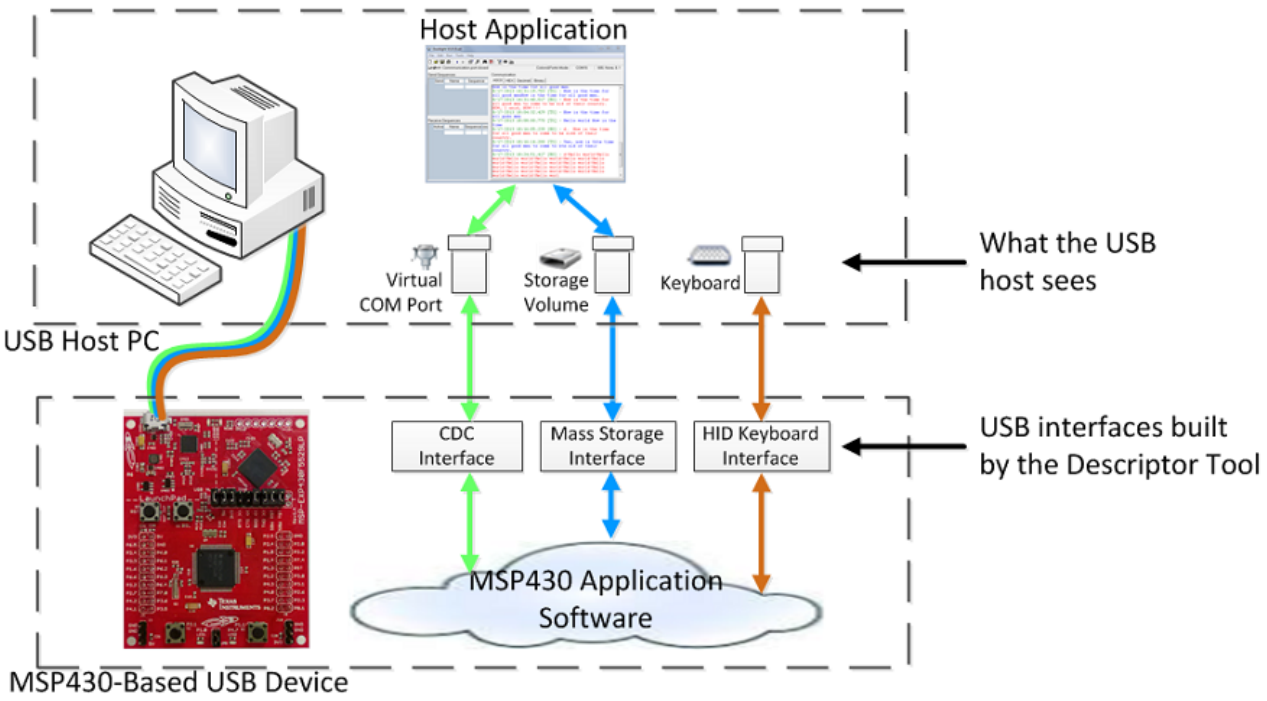


Figure 4. The Descriptor Tool

In addition to USB descriptors, the Tool collects other compile-time information related to resources owned by the USB API, so that the API can manage them properly. (See Table 5.)

Once the Tool's output is generated and placed into the target project, the developer can immediately begin writing application-specific code that sends/receives data over USB. The provided examples are a good starting point for this.

5.2 What is a "USB Interface"?

A USB interface is essentially a partitioned stream of data between the USB device and the host. An interface is of a particular device class, each with its own protocol. The host responds differently for each device class:

CDC (Comms Device Class): Presents a virtual COM port
HID (Human Interface Device): Subtypes include mice, keyboards, other PC peripherals, or simply "generic".
MSC (Mass Storage Class): Causes the host to mount a storage volume over this interface
PHDC (Personal Healthcare Device Class): Only used with Continua medical devices

The device tells the host what interfaces it contains by reporting the information in the *USB descriptors*. Most devices contain only a single interface, but some contain more; these are called composite USB devices.

The Descriptor Tool can quickly build descriptors for any combination of CDC, HID, and MSC interfaces, according to whatever your application needs.

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. In the MSP430 USB API, 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.)

5.4 When to Use the Tool?

Before you can write USB code, you need to create your USB interfaces with the Tool. So, run the Tool, before beginning your development.

When you run the Tool, be sure to save your inputs into a *.dat file. This way, you can make changes and re-generate your output later. This is often necessary; for example, a products VID/PID often aren't finalized until later in the project.

5.5 Using the Tool

At the beginning of development, users are often most concerned about setting up their USB interfaces and starting development. Before sending a device or product into the field, however, every field needs to be fine-tuned.

Both approaches are described below.

While using the Tool, always check the Help Pane text for each field – it helps you understand the tradeoffs of the decision you need to make. Float over each field, or click on it, to update the Help pane.

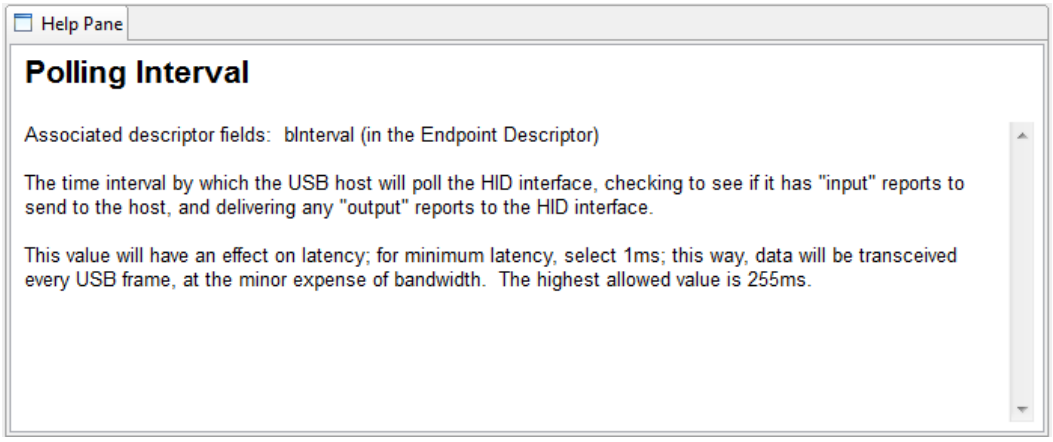


Figure 5. The Descriptor Tool's Help Pane

5.5.1 Minimum Steps to Begin Development

These are the minimum steps required to build interfaces, to start development.

1) Select your interfaces, by pressing the buttons at the top. Before this step, you will have needed to decide on your interfaces; read more about them in this Programmer's Guide.

Figure 1. Selecting Your USB Interfaces

Upon doing this, the interfaces will appear in the Navigation View, at left.

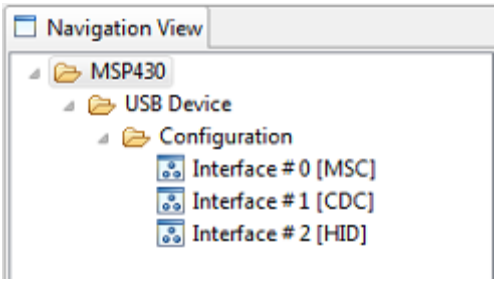


Figure 2. Interfaces in the Navigation View

The Tool catches certain situations that might have caused you trouble, and warns or corrects you about them. An example of this is that it will always move an MSC interface to the front of the list; this maximizes compatibility with Windows/Mac/Linux operating systems.

2) With the tree formed, now click on each node, filling out relevant information in each node's pane. The items below are the minimum requirements to create the interfaces.

a) **MSP430 View --> XT2 Oscillator Frequency:** Select the frequency to match the clock you're applying on your MSP430's XT2 oscillator. If this doesn't match your hardware's oscillator, your USB won't function.

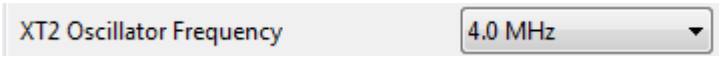


Figure 3. XT2 Oscillator Frequency Field

b) **USB Device View --> PID:** The VID field is already filled out with TI-MSP430's VID, but you must enter a PID. The Help Pane text includes an "experimenter's range" of PIDs you can use.

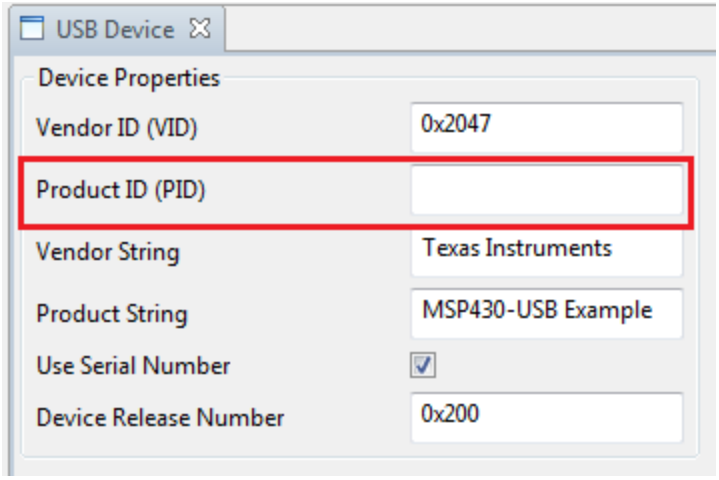


Figure 4. PID Field

c) **HID interfaces:** choose your device type (datapipe, mouse, keyboard, or custom)

d) **MSC interfaces:** choose the number of LUNs and whether the media is removable

For c) and d), reference the Help Pane text for these fields, as well as this Programmer's Guide.

3) Save your *.dat input file. (File --> Save As...). This way, you can load your inputs at a later time, modify, and re-generate. (The Tool cannot read descriptors.c/h output files, only the *.dat file.)

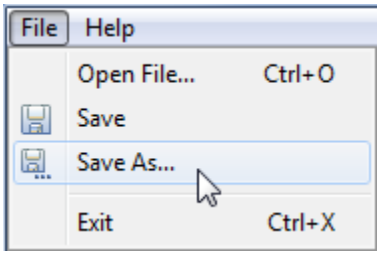


Figure 5. Save As…

4) Generate your output, by pressing the "Generate" button, on the far right, which looks like a "gear". Direct the output to your target project. In the typical project structure for the USB API, these files are usually stored in \USB_config (see the example projects). But they can go anywhere, as long as the project's #include settings are adjusted accordingly.

Figure 6. The "Generate" Button

6) Build your project. It's recommended to use the "Rebuild all" option the first time compiling after replacing the default files with Descriptor Tool output files.

5.5.2 Using the Tool Before Final Production

The items above tell the host what kind of device it is, and help configure the MSP430's minimum information to function as a USB device. But the fields that were skipped over are important.

Some of them may even be needed prior to development for some projects, like the one indicating the project will be used with the SYS/BIOS RTOS. Other fields, like the vendor/product/configuration/interface strings, are used by the host to identify the USB device.

To do a complete job of using the Tool, follow the same procedure as above, but pay attention to every field. Use the Help Pane to understand the decisions you're making. If the information there isn't sufficient, please reference relevant portions of this Programmer's Guide.

5.6 The Tool's Generated Output

The Tool outputs four files:

- *descriptors.c*
- *descriptors.h*
- *usbisr.c* (contains the USB interrupt service routine, and related functions)
- an INF file (if one of the interfaces is CDC)

When a CDC interface is attached for the first time to a particular Windows host, Windows needs the user to provide an INF file (*.inf). Windows uses INF files to associate devices with particular drivers, and for HID/MSC interfaces, the INF is included in Windows. For CDC it isn't. The INF contains specific information about the device, including its VID/PID and some strings.

Since the Tool already has this information, it generates an INF file for you. Also note that if your USB device is composite, the INF file must change according to the interfaces you selected. The Tool handles this for you, automatically.

A diagram of how the Tool configures the API is shown below.

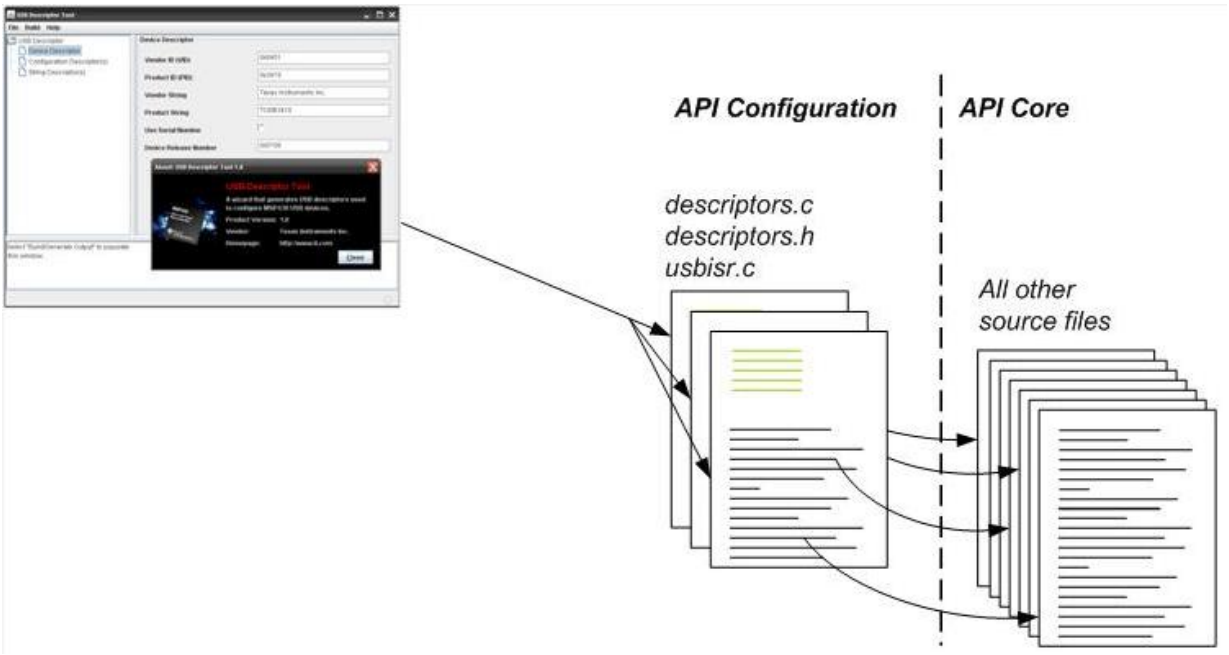


Figure 7. Descriptor Tool: File Interaction

5.7 Accessing Interfaces from the Application

For USB devices including multiple CDC or HID interfaces, it can be important for the host and device to agree on how the interfaces are mapped, so that they can send/receive data on the appropriate interface.

intfNum is the index you provide to API function calls on the MSP430 side, to select the appropriate interface. In the Descriptor Tool, every USB interface has an *intfNum* parameter. This is seen on each interface's view:

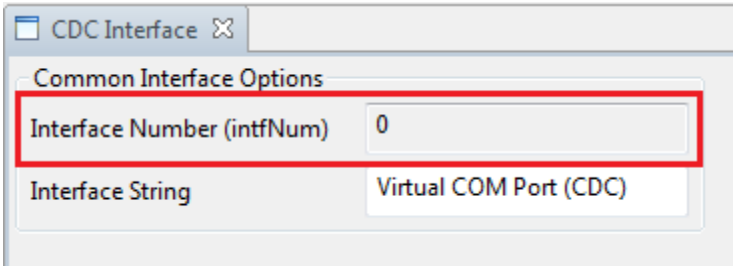


Figure 8. The intfNum Parameter

For every interface you define, the Tool places a constant in the descriptors.h file, like CDC0_INTFNUM or HID0_INTFNUM. You can use these as the intfNum parameter in your application code:

```
// Sends data to the CDC interface with intfNum=1, the second CDC
// interface defined in this device
cdcSendDataInBackground(buf, size, CDC1_INTFNUM);
```

For single-interface (non-composite) devices, the sole interface's intfNum is always zero.

There can only be one MSC interface in a USB device, so there is no intfNum parameter in the MSP430's MSC API calls. If MSC is used in composite, it's always reported as the first interface in the device, per USB requirements.

5.8 USB Configurations

The USB specification provides for multiple *USB configurations* to be defined within a given USB descriptor set. Each configuration contains a set of USB interfaces that are in effect at any given time, and the host theoretically has the option to change configurations at any time.

So for example, the device could report two configurations: one with an HID keyboard interface, and another with both a keyboard and mouse HID interfaces. The host could choose to switch between those two interfaces, almost like selecting two different logical devices within the same physical device.

However, in practice, this feature doesn't see much use. The MSP430 USB API only supports definition of a single configuration within its descriptor set.

5.9 Trouble Opening the Tool?

The Tool requires Java Runtime Engine v1.5+ to run properly. As a result, you may need to download the most recent version of the JRE. The Tool is designed to automatically sense the JRE version on your machine upon being opened, and guide you to http://www.java.com if necessary.

Due to Java limitations, the Tool may give an error upon being opened if there is a semicolon or space in the pathname, or if the pathname is very long. If the Tool fails to open, with an unclear error, please move the entire directory for the tool into a different path that addresses these two problems.

6 USB States/Event Management

6.1 USB Management Function Calls

The USB API uses the following function calls to manage basic USB states. Related USB event handlers are also shown.

Table 7. USB Management Call Summary

Function	Description
`BYTE USB_setup()`	Intended to be called at the beginning of execution. • Calls `USB_init()` • Calls `USB_setEnabledEvents()` to enable all events • If a USB host is found to be already present, calls `USB_reset()`, `USB_enable()`, and `USB_connect()` to prompt enumeration.
`BYTE USB_init()`	Prepares the USB module to detect VBUS events from the host. (Calling this from the application has been deprecated; use `USB_setup()` instead.)
`BYTE USB_enable()`	Activates the USB module, PLL, and transceiver (PHY).
`BYTE USB_enable_PLL()`	Special functions used only if the developer has decided to reduce USB resume interrupt latency, by assuming responsibility for oscillator/PLL delay periods. USB_enable_PLL() starts the PLL lock after the application has ensured that XT2 has stabilized. USB_enable_final() finishes the resume, after the application has ensured the PLL lock period has expired. See USB example #G1.
`BYTE USB_enable_final()`	
`BYTE USB_disable()`	Disables USB module, PLL, and transceiver (PHY).
`BYTE USB_setEnabledEvents()`	Enables/disables detection of the available events.
`BYTE USB_getEnabledEvents()`	Returns the status of USB event enabling
`BYTE USB_connect()`	Instructs USB module to make itself available to the host for enumeration, by pulling the PUR pin high.
`BYTE USB_disconnect()`	Causes a logical disconnect from the host by pulling PUR 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

Table 8. USB Event Handler Summary

Event Handler functions	Event Description
`BYTE USB_handleClockEvent()`	The USB PLL has failed (out of range). This may also mean the XT2 crystal oscillator has failed.
`BYTE USB_handleVbusOnEvent()`	Indicates that a valid voltage is now available on the VBUS pin (a host is now present on the bus)
`BYTE USB_handleVbusOffEvent()`	Indicates that a valid voltage is no longer available on the VBUS pin (a host was present on the bus, but has now been removed)
`BYTE USB_handleResetEvent()`	Indicates that the USB host has initiated a port reset. This has a similar effect to calling `USB_reset()`, including that enumeration (if it had been achieved) will be lost, and the USB state will change.

`BYTE USB_handleSuspendEvent()`	Indicates that the USB host has suspended this device.
`BYTE USB_handleResumeEvent()`	Indicates that the USB host has resumed this device from suspend mode.
`BYTE USB_handleEnumCompleteEvent()`	Indicates that the USB device has just become enumerated.
`BYTE USB_handleCrystalStarted()`	Special functions used only if the developer has decided to reduce USB resume interrupt latency, by assuming responsibility for oscillator/PLL delay periods. `USB_handleCrystalStarted`() indicates the API has started the stabilization process, and the application should take responsibility to finish it and then call `USB_enable_PLL()`. `USB_handlePLLStarted()` indicates the API has started the PLL lock process, and the application should wait the specified delay and then call `USB_enable_final()`. See USB example #G1.
`BYTE USB_handlePLLStarted()`	

For reference details on these functions, including description, parameters, and calling conditions, please see the API Functional Reference within the USB Developers Package. The functions are further described below.

6.2 USB States/Events

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.

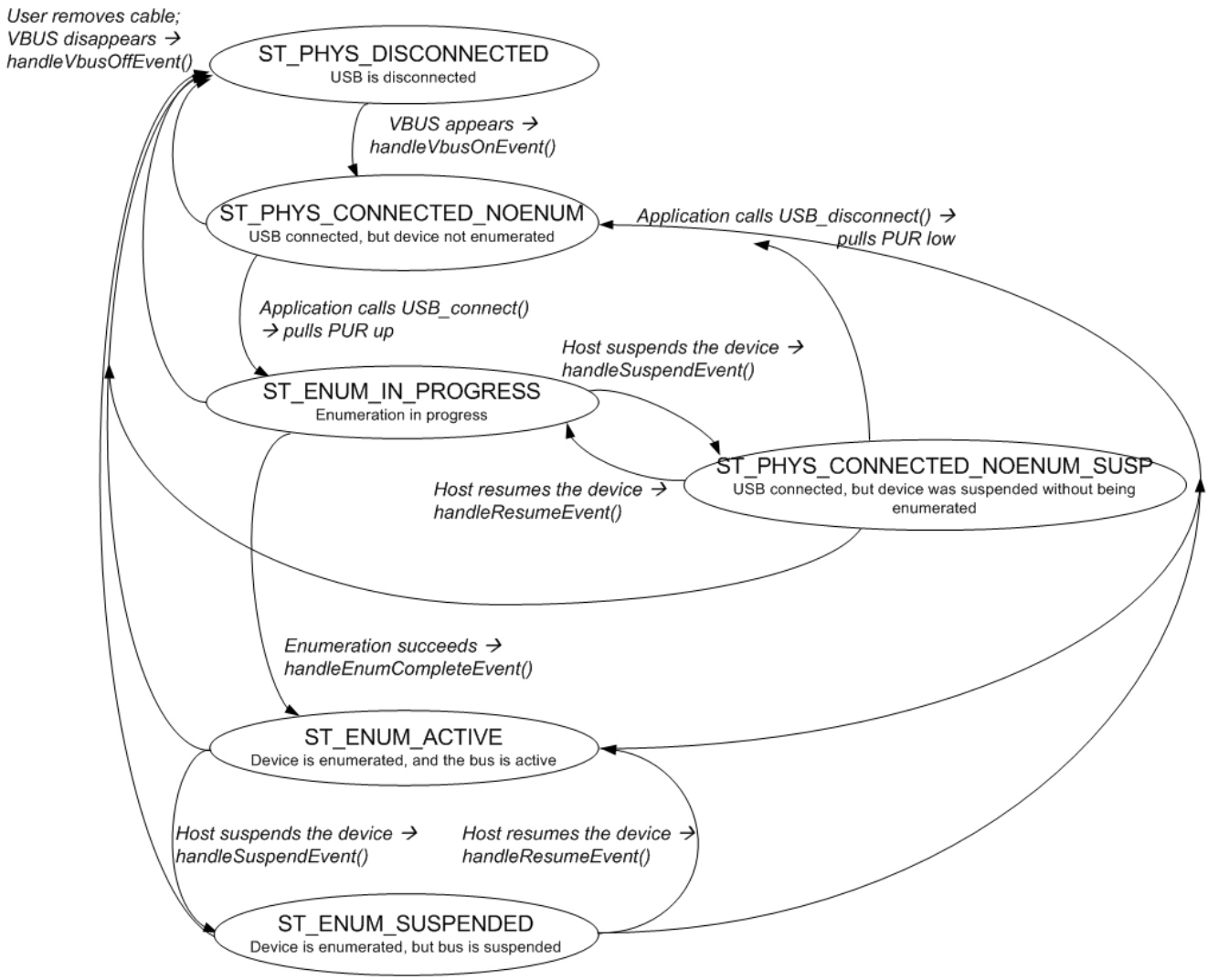


Figure 9. USB Device State Diagram

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 9. USB State Definitions

	USB_connectionInfo()				
USB_connectionState()		VBUS detected?	PUR high?	Enumerated?	Suspended?
	ST_PHYS_DISCONNECTED				
	ST_PHYS_CONNECTED_NOENUM	X			
	ST_ENUM_IN_PROGRESS	X	X		
	ST_ENUM_ACTIVE	X	X	X	
	ST_ENUM_SUSPENDED	X	X	X	X
	ST_PHYS_CONNECTED_NOENUM_SUSP	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 usually 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

Many applications are best served by using `USB_connectionState()` to direct program flow.

The states are reflected in the suggested main loop framework shown below. This same framework is located within the empty USB project.

```
void main (void)
{
    // Set up clocks/IOs.  initPorts()/initClocks() will need to be customized
    // for your application, but MCLK should be between 4-25MHz.  Using the
    // DCO/FLL for MCLK is recommended, instead of the crystal.  For examples
    // of these functions, see the complete USB code examples.  Also see the
    // Programmer's Guide for discussion on clocks/power.
    WDT_A_hold(WDT_A_BASE);  // Stop watchdog, so it doesn't reset your app
    initPorts();             // Configure all GPIOs
    initClocks();            // Configure clocks

    // Initialize the USB module, and connect to the USB host (if one is present)
    USB_setup(TRUE, TRUE);

    __enable_interrupt();    // Enable general interrupts

    while (1)
    {
      // This switch() creates separate main loops, depending on whether USB
      // is enumerated and active on the host, or disconnected/suspended.  If
      // you prefer, you can eliminate the switch, and just call USB_connectionState()
      // prior to sending data (to ensure the state is ST_ENUM_ACTIVE).
      switch(USB_connectionState())
      {
        // This case is executed while your device is connected to the USB
        // host, enumerated, and communication is active.  Never enter LPM3/4/5
        // in this mode; the MCU must be active or LPM0 during USB communication
        case ST_ENUM_ACTIVE:

        // These cases are executed while your device is:
        case ST_PHYS_DISCONNECTED: // physically disconnected from the host
        case ST_ENUM_SUSPENDED:   // connected/enumerated, but suspended
        case ST_PHYS_CONNECTED_NOENUM_SUSP: // connected, enum started, but host unresponsive

            // In this example, for all of these states we enter LPM3.  If
            // the host performs a "USB resume" from suspend, the CPU will
            // automatically wake.  Other events can also wake the CPU, if
            // their event handlers in eventHandlers.c are configured to return TRUE.
            __bis_SR_register(LPM3_bits + GIE);
            break;

         // The default is executed for the momentary state
         // ST_ENUM_IN_PROGRESS.  Almost always, this state lasts no more than a
        // few seconds.  Be sure not to enter LPM3 in this state; USB
        // communication is taking place, so mode must be LPM0 or active.
         default:;
       }
   }  //while(1)
} //main()
```

This framework is used in most of the examples, and described in more detail in Sec. 11.8.

A device is likely to spend most of its time in the disconnected, active, and suspended states.

6.3 Initializing the API

Before any other API call, the application must initialize the USB module. Prior to v4.0 of the USB API, this was done with a call to `USB_init()`. Events were then enabled using `USB_setEnabledEvents()`.

These functions still exist and are used internal to the API, but for the USB API v4.0 and later, TI now recommends most application use a new function instead: `USB_setup()`.

This function:

- calls `USB_init()`
- optionally enables all events (selectable by its first parameter)
- optionally performs an immediate `USB_connect()` if it finds that a physical host connection is present. (selectable by its second parameter)

These behaviors are common to the vast majority of applications, allowing all of them to be handled within a single function call. If there is a desire to control individual event enables, the old method can still be used.

`USB_setup(TRUE, TRUE)` can be seen in the code segment shown in the previous section.

6.4 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: A couple of unusual situations may result in 5V on the VBUS pin without a valid host. One is if the device is attached to a self-powered hub without a host being present. Another is if the host is powered, but has become hung. A USB device has no way of distinguishing these from an active host, and will generally assert PUR to attempt enumeration (for example, call `USB_connect()`). If this is attempted, the USB state will fail to enter `ST_ENUM_ACTIVE`, and instead will enter `ST_PHYS_CONNECTED_NOENUM_SUSP` indefinitely. The application may need to take this into account.

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

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 (see below).

6.5 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();
}
```

Most of the USB API examples include this code in `handleVbusOnEvent()`.

Note that `USB_setup()` can perform an automatic connection, if instructed to by its parameters, and if it finds VBUS already present. This is useful for initial startup of execution, where if VBUS is already present, no VBUS-on event will be seen.

6.6 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_PHYS_DISCONNECTED` and a `handleVbusOffEvent()` is generated), or when a call is made to `USB_disconnect()` or `USB_disable()`.

6.7 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.

Although the above actions are taken automatically, the application developer is responsible for ensuring that the device doesn't consume more power than it should from the host VBUS line. This may occur if the MSP430 DVCC or the rest of the board is powered from VBUS. See Sec. 11.1 for more information.

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.

6.8 Selective Suspend

The description above applies to USB suspend in general. Traditionally, all USB devices were suspended at the same time. In contrast, *selective suspend* is a newer feature by which the host may suspend individual USB devices selectively, without suspending the others. Usually this would be done after the host decided that communication had been idle for a period of time. Use of selective suspend has grown in recent years, mostly in the realm of mobile devices like phones and tablets.

From the USB device standpoint (i.e., MSP430), there is no difference between a general suspend or selective suspend. In either case, what the MSP430 sees is 3ms of inactivity, and it responds the same way. Thus, it supports selective suspend.

Having said this, the USB application may have some impact on whether the USB host considers the device "inactive", and thus whether it selectively suspends the device. .Also, Windows requires a device to have remoted itself as having remote wakeup functionality, before it will consider it for selective suspend. Sec. 11.1.5 for additional information on optimizing selective suspend.

6.9 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.

6.10 Failed Enumeration

If the device is attached to a host, and software calls `USB_connect()`, the host usually begins enumerating it immediately. The state shifts to `ST_ENUM_IN_PROGRESS`. Usually enumeration finishes quickly, and the state moves to `USB_ENUM_ACTIVE`.

However, for a variety of reasons, this may not happen:

- the device becomes physically attached to a powered hub that doesn't have a host upstream from it. It sees VBUS, and interprets this as the presence of a host, and calls `USB_connect()`. But there is no host, and so the bus remains idle.
- the device becomes physically attached to a host that is indeed powered, but in a 'standby' mode (during which all USB devices are suspended). It sees VBUS and calls `USB_connect()`. But because the host is in standby, the bus remains idle. (In other words, the host suspends the device, just like all USB devices that were already attached.)
- the device becomes physically attached to a host that is indeed powered, but hung; i.e., "blue screened", or unusually busy. It sees VBUS and calls `USB_connect()`. But because the host is hung, the bus remains idle, or the process otherwise doesn't complete.
- the device becomes physically attached to a host, and the host begins enumeration. But something goes wrong in the process; this could be the fault of either the host or device. The host gives up and suspends the device.

USB suspend is defined as >3ms of no activity from the host. Therefore if the host fails to finish (or even start) the process, and instead the bus goes idle, then the device's state will shift from `ST_ENUM_IN_PROGRESS` to `ST_PHYS_CONNECTED_NOENUM_SUSP`. It will remain there until the host finishes enumeration, or until software "cancels" it (using `USB_disconnect()` or `USB_disable()`).

Since the events listed above are very possible in the lifetime of the USB device, it's a good idea for the software developer to consider what will happen in the `ST_PHYS_CONNECTED_NOENUM_SUSP` state. Software could choose to stay in this state until the bus situation changes. If so, it remains ready for the host to enumerate it at any time. MSP430 low-power modes down to LPM3 are allowed in this state.

Alternatively, software could choose to call `USB_disconnect()`, and/or `USB_disable()`, shifting the state to `ST_PHYS_CONNECTED_NOENUM`. However, there's no obvious advantage to this; and the developer should be aware that if the host does later become available while the device is attached to it, no event will be received by the device to make it aware of this and re-connect. Therefore the user will probably need to detach/re-attach before enumeration will take place.

6.11 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 automatically disabling the USB module. It also calls `handleVbusOffEvent()`, where application software might be written to take action. After the VBUS-off event, a call to `USB_connectionState()` will return `ST_PHYS_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.

6.12 USB Hardware Conditions in Each State

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

Table 10. USB State Hardware Conditions

	USB Module Enabled	USB Transceiver[1]	Integrated LDOs[2]	XT2 Oscillator	PLL Enabled
ST_PHYS_DISCONNECTED					
ST_PHYS_CONNECTED_NOENUM	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_PHYS_CONNECTED_NOENUM_SUSP	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, thus it must be enabled in the majority of states. 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 Data Exchange using Datapipe Interfaces (CDC and HID-Datapipe)

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

This section is a reference guide for how datapipe interfaces work. For practical usage information, see Sec. 11.10 and Sec. 11.11.

7.1 Introduction

The MSP430 USB 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*. It defines an architecture in which the developer's MSP430 application can define a continuous data block at any address and of any size, and send it to the host; or receive a block of any size from the host, to be deposited at any particular address. In this sense, it's much like interfacing with a COM port or UART, freeing the developer from thinking about the USB plumbing.

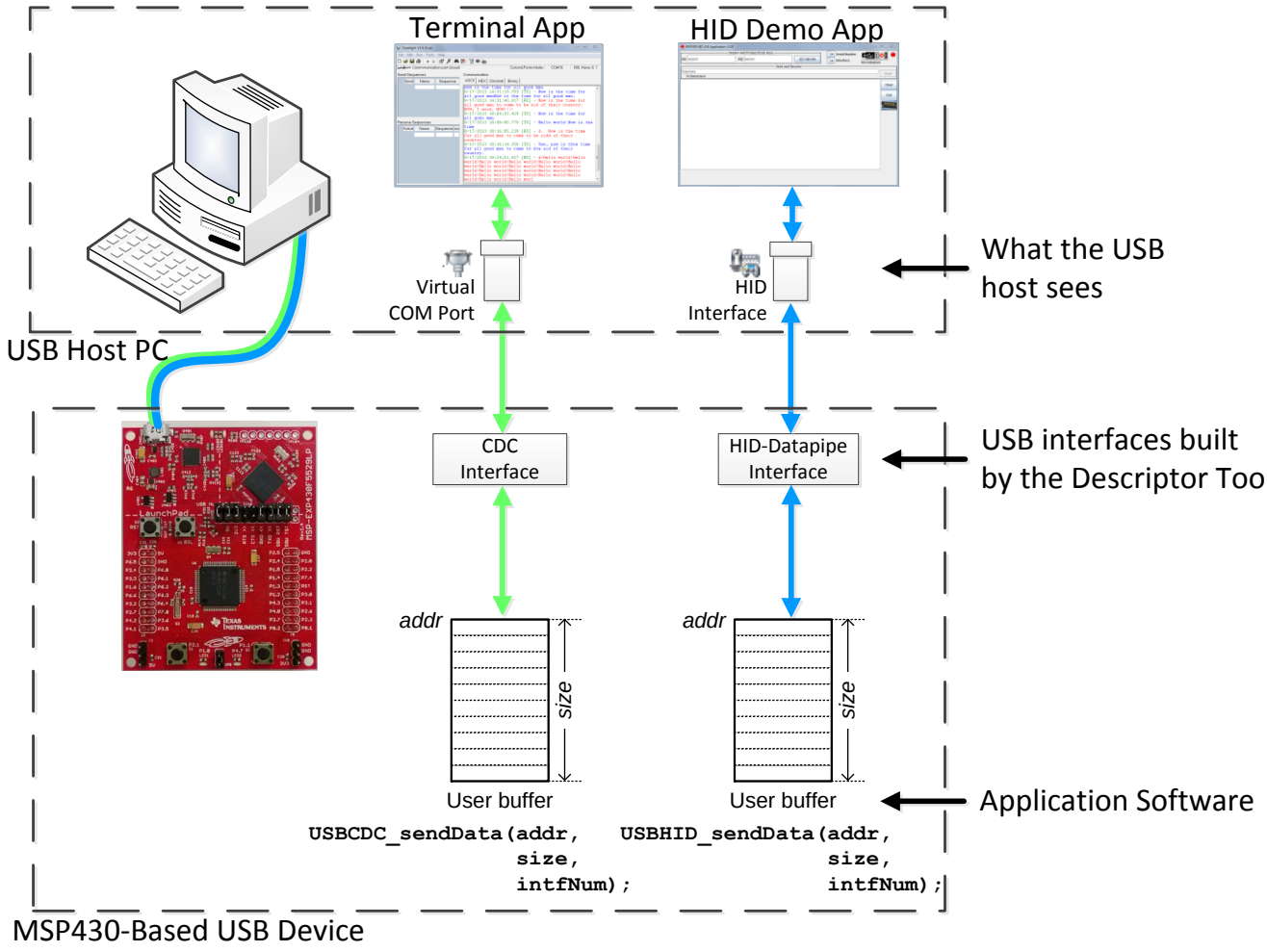


Figure 10. Datapipe Interfaces

All CDC interfaces in this API are datapipe interfaces. HID interfaces in the MSP430 USB API can be either traditional or datapipe. This section does not apply to HID-Traditional devices, like mice or keyboards.

HID-Datapipe was created because HID has some advantages over CDC; for example, the fact that HID devices enumerate silently on the host, without an INF file. But HID-Traditional is more difficult to develop; thus HID-Datapipe offers the best of both world (at the expense of throughput, limited at 64KB/sec).

Either way, the datapipe interface looks exactly the same. The calls are symmetrical, differing only in their prefix (`USBCDC_` vs. `USBHID_`). Since the text in this discussion applies equally to both, the function calls are sometimes shown to have the prefix `USBxxx_`. The same applies to the prefix for return values (`kUSBxxx_`).

7.2 Datapipe Function Call / Event Summary

A table summarizing the datapipe function calls is shown below. For complete documentation, see the API call reference within the USB Developers Package.

Table 11. Datapipe Function Call Summary

Function	Description
`BYTE USBCDC_sendData()` `BYTE USBHID_sendData()`	Begins a send operation to the USB host
`BYTE USBCDC_receiveData()` `BYTE USBHID_receiveData()`	Begins a receive operation from the USB host
`BYTE USBCDC_bytesInUSBBuffer()` `BYTE USBHID_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()` `BYTE USBHID_abortSend()`	Aborts an active send operation
`BYTE USBCDC_abortReceive()` `BYTE USBHID_abortReceive()`	Aborts an active receive operation
`BYTE USBCDC_rejectData()` `BYTE USBHID_rejectData()`	Rejects payload data residing in the USB buffer, for which a receive operation has not yet been initiated
`BYTE USBCDC_intfStatus()` `BYTE USBHID_intfStatus()`	Returns status information specific to a particular datapipe interface

Table 12. Datapipe Event Handlers

`BYTE USBCDC_handleDataReceived()` `BYTE USBHID_handleDataReceived()`	Indicates that data has been received for CDC or HID interface `intfNum`, for which no data receive operation is underway
`BYTE USBCDC_handleSendCompleted()` `BYTE USBHID_handleSendCompleted()`	Indicates that a send operation on CDC or HID interface `intfNum` has just been completed
`BYTE USBCDC_handleReceiveCompleted()` `BYTE USBHID_handleReceiveCompleted()`	Indicates that a receive operation on CDC or HID interface `intfNum` has just been completed

For reference details on these functions, including description, parameters, and calling conditions, please see the API Functional Reference within the USB Developers Package. The functions are further described below.

7.3 Creating a Datapipe Interface

It's recommended to use the Descriptor Tool to create the device's interfaces. Any CDC interface created by the Tool is a datapipe interface, so all the CDC function calls are datapipe calls. To create an HID-Datapipe interface, add an HID interface in the Tool, and in its "Interface" pane, select "HID-Datapipe". Then after generating the interfaces, interact with this interface using the HID-Datapipe function calls.

HID-Datapipe interfaces are distinguished from HID-Traditional in two main ways:

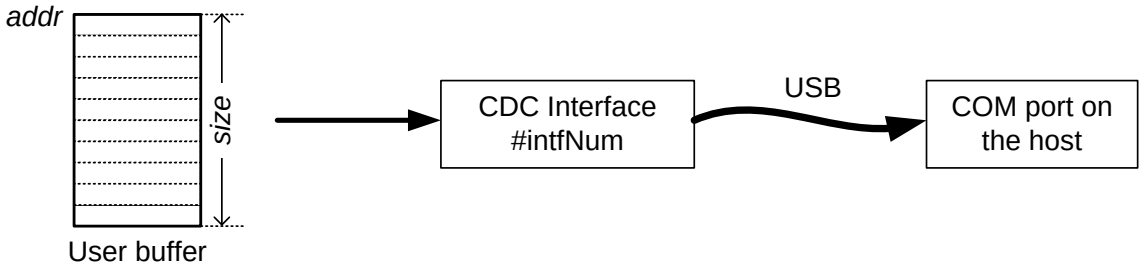
- The HID report format is different. HID-Datapipe defines a report format with a single byte field, and the remainder defined as a large data field. Unlike most HID reports, it doesn't attempt to define an intricate report structure.
- The use of the HID-Datapipe calls instead of the HID-Traditional calls. (for example, `USBHID_sendData()` vs. HID-Traditional's `HID_sendReport()`). The HID-Datapipe calls assume the aforementioned report format.

7.4 Send/Receive "Operations"

The basic unit of all sending/receiving in the datapipe is the *operation.* Sending data requires the start of a *send operation.* Receiving data requires the start of a *receive operation.*

For all sending and receiving, the application must first prepare a data source/sink, called the *user buffer*. The application passes the buffer, the number of bytes to be transceived, and the desired USB interface (index with `intfNum`) to `USBxxx_sendData()` or `USBxxx_receiveData()`. The user buffer is any contiguous block within the MSP430 memory map, described by an address `addr` and size `size`.

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




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

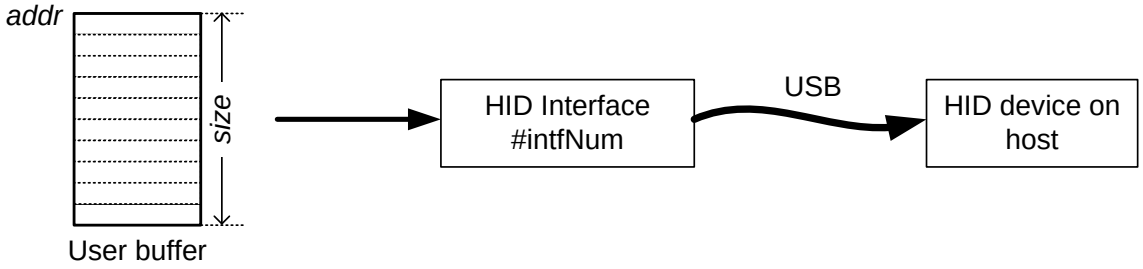


Figure 11. Send Operation

A single call to `USBxxx_sendData()` initiates a *send operation.* The API begins to copy the buffer to the USB endpoint buffer (128 bytes), which is made available to the host. At the host's discretion and timing, it reads the packet from the MSP430's endpoint buffers. As the endpoint buffer is emptied, the API fills it with the next data from the user buffer. When all the data has been read by the host, 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 over this interface. 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.

7.4.1 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 to a small 64 bytes. The API handles all interactions with the endpoint buffers automatically, so they're not part of the API application interface 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.

7.4.2 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, in that it happens asynchronous to the application's code execution, driven by an alternate interrupt source (the host). Another way of looking at it is that the USB MCU's execution is "decoupled" from the host, rather than being dependent on it for a quick response.

Background execution 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.

When the MSP430 software starts (for example) a send operation using `USBCDC_sendData()`, it prepares communication with the host, and then waits for the host to read all the USB packets necessary to complete the operation. (USB is a very host-centric bus, and no byte can be sent/received without the host requesting it.) Then, possibly even before a byte of data is transferred, `USBCDC_sendData()` returns, and the application execution continues.

As the host begins pulling the data, interrupts occur in the MSP430 to fulfill the operation. The application is not mindful of this. The API 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 – the API calls `USBxxx_handleSendCompleted()` to inform the application. The interface is once again available for a new operation.

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()`.

Note that the developer can effectively avoid background execution (or in the terminology used in other software contexts, achieve *synchronous* operation), by polling the interface with `USBxxx_intfStatus()` before allowing application execution to resume. The API provides two example constructs for each case, for example `hidSendDataInBackground()` and `hidSendDataWaitTilDone()`, which implement both approaches – see Sec. 11.10.

7.4.3 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 exists, each can have their own simultaneous operations.

7.4.4 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.

7.4.5 Lifecycle of a Send Operation

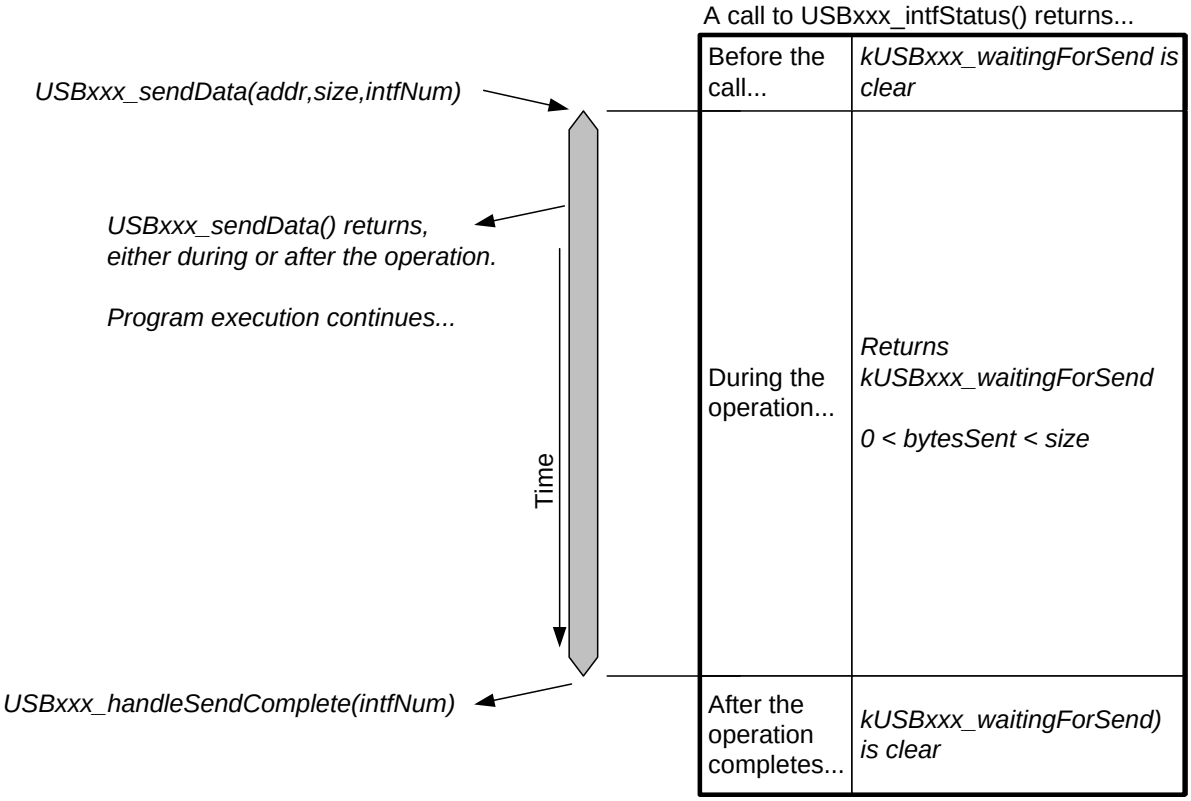


Figure 12. Lifecycle of a Successful Send Operation

As discussed earlier, a send operation is begun with a call to `USBxxx_sendData()`. The application can make its first call to this function at 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 remember that this is dependent on the host and bus conditions. 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.

7.4.6 Lifecycle of a Receive Operation

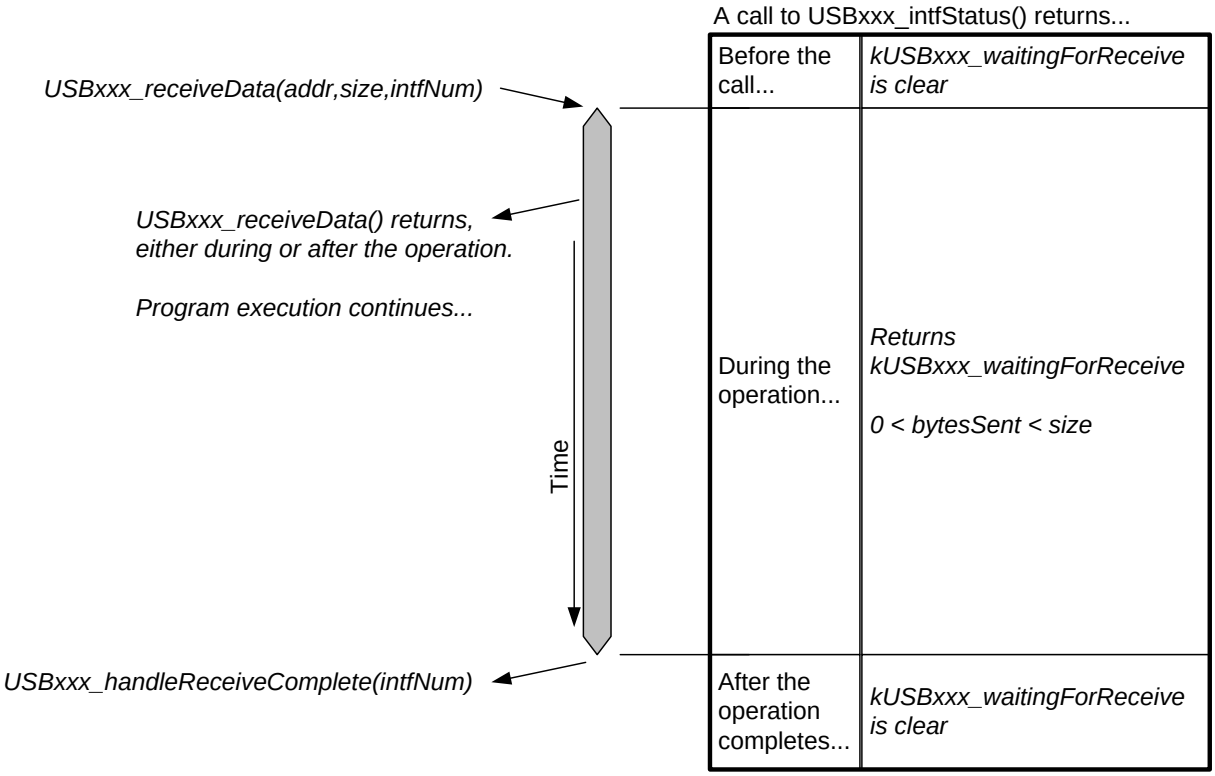


Figure 13. Successful Receive Operation

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.

7.4.7 How Long (in Real-Time) 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 the developer controls both the host and device, and/or 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. 11 provides clear example coding constructs that ensure proper operation.

7.5 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.

The downside of CDC, on a Windows host, is that an INF file is required, and Microsoft has made it difficult to install this without it being signed. On Linux/Mac, CDC enumerates silently as a virtual COM port. (See Sec. 2.2.2 for more discussion.)

With HID, the host situation is reversed. An HID interface (datapipe or otherwise) enumerates silently on Windows/Mac/Linux hosts. However, the application end is a little more complicated. HID is mostly unique to USB, and therefore the knowledge of how to interface with it on the host is less common. Writing an HID application requires defining a report format. To interface with HID-Datapipe, the application will need to specify the report format used by the MSP430 HID-Datapipe interface, shown below.

Table 13. 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 usually-complicated HID report mechanism into a simple data carrier.

The MSP430 application sees only an unformatted stream of data. The *Java HID Demo App*, provided within the MSP430 USB Developer's Package, does the same thing for any host equipped with a Java Runtime Engine.

8 MSC: Software Architecture

The mass storage class (MSC) defines an interface through which the USB host expects to mount a storage volume. MSC essentially acts as a carrier for industry-standard SCSI commands. After the host enumerates/installs an MSC interface, it uses SCSI commands to attempt to mount a storage volume.

This section explains how to implement an MSC interface using the MSP430 USB 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 provides a large ("mass") pool of data – a *storage volume*, written on a *medium* – on which the host can "read" and "write". In an MCU-based application, the medium might be the MCU's internal memory, or an external device accessible via an externally-facing interface (for example, SPI, I2C, or parallel memory interface).

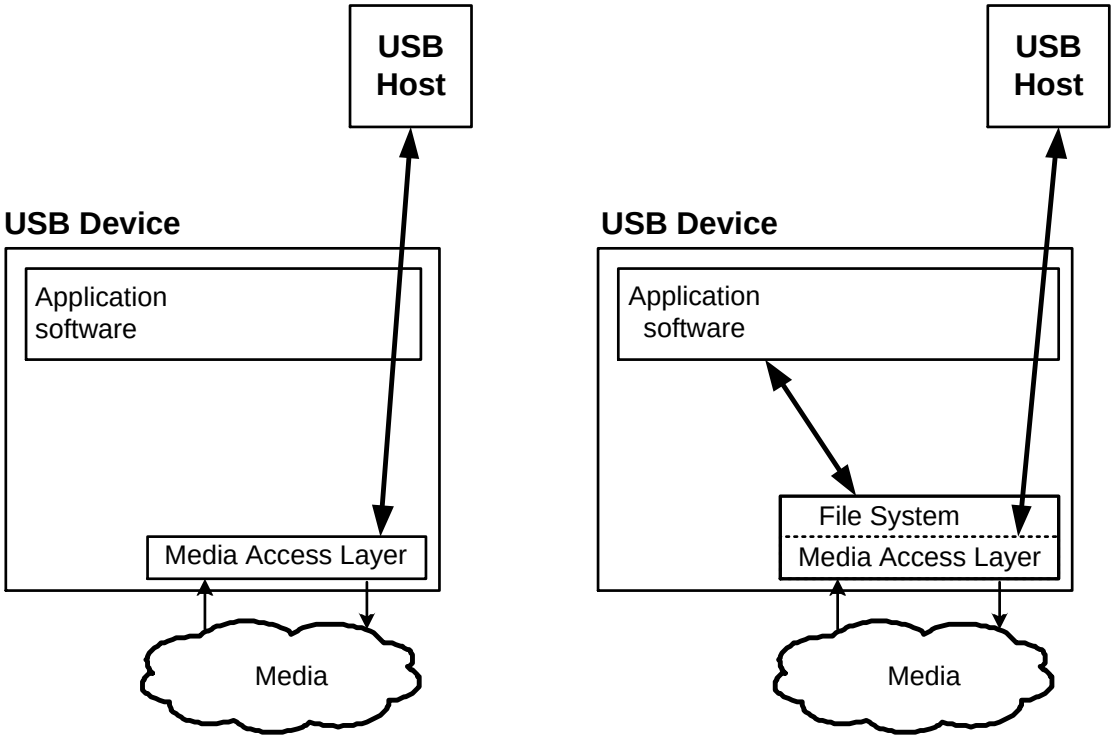


Figure 14. Simplified High-Level MSC Architecture

Whatever the medium, the host expects this volume to be formatted in a manner consistent with a standard storage volume format it understands, such as the common FAT (File Allocation Table) standard. FAT has been used since MS-DOS days, and is still popular for smaller volumes based on random-access media. Support for it is universal among the major operating systems.

Compared to CDC and HID, in which the application's task is simply to send/receive data, MSC applications have more work to do. The application is responsible for acting upon any READ/WRITE commands received from the host, which is receives from the API. This work is heavily defined by the chosen media -- which is why the API is limited in the extent to which it can automate it. If the application doesn't faithfully access the volume at the host's request, then although the MSC interface can be seen as enumerated on the host, the volume won't mount.

The examples accompanying the USB Developer's Package include implementations for common media, like SD-cards (#M2).

In Figure 14, the image at left shows an implementation in which the MSP430 application doesn't need access to the volume's data for its own purposes. It's able to carry out the low-level read/write requests from the host; but since it can't parse the volume's format, it can't find any particular file on the volume. In this sense, it's essentially acting as a "flash drive".

The image at right shows the more typical case where the MSP430 application does need to find files on the volume. To accomplish this, it uses file system software capable of parsing the volume's formatting. Through this software, it's able to access files by directory and filename. All block requests from the host are still passed through the application, this time to the file system's low-level media access functions.

A popular open-source software solution for the FAT storage is FatFs. Many of the MSC application examples include an MSP430 port of FatFs.

Another example, #M1, works around the need for file system software by using "file system emulation". This approach solves the problem of the application finding files, by restricting where the host can put files. However, it imposes limits on the host with respect to how many files can be stored, and where.

8.2 MSC Device Types

MSC devices must report a media type to the host. The MSP430 MSC API supports two types, shown in the table below.

Table 14. MSC Device Types Supported by the MSP430 USB API

Name	Common Examples	SCSI Command Set Used	File System Typically Used
Direct-Access Block Devices	Hard drives, flash drives, SD-card, MultiMedia Card. Most forms of random-access, magnetic/flash-based media.	SBC-2	FAT16/32 (File Allocation Table)
CD/DVD	CD-ROM drives; DVD drives	MMC-4	ISO9660 (CD File System)

The determination of which type is reported to the host is made by whether the configuration constant `CDROM_SUPPORT` is defined. If it is not defined, then the device is reported as a direct-access block device. The Descriptor Tool automatically defines this constant (or not), based on settings chosen within the Tool.

Although the MSP430 is capable of being used as the USB interface for a CD-ROM drive, the purpose of the CDROM function is more about CD-ROM emulation. Files on a CD-ROM can be "autorun" executed, and this works on multiple host operating system platforms, unlike a direct-access block device.

To accomplish this emulation, the media must be pre-formatted as an ISO image. This is sometimes emulated as a file located on a FAT-formatted volume. Example #M5 shows this kind of implementation.

Since FAT-based direct-access block devices are much more common, most of the forthcoming discussion, as well as the examples, are written from that perspective. However, the API is intended to be flexible enough to adapt to other file formatting systems.

8.3 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 also 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, which then accesses the volume on the medium. LBAs begin at zero.

Usually the LBA gets passed as a parameter to the MSP430 application's file system software, as the target location for READ/WRITE. Usually the number of blocks is also passed (starting at the given LBA).

Alternatively, if the MSP430 application doesn't need to interpret the volume for its own use, the application might choose to convert the LBA to a byte address within the target medium, by multiplying the LBA by the size (in bytes) per logical block, or sector; then add this to any appropriate offset within flash.

The LBA has meaning 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* (keeping in mind that only one MSC interface is allowed, within a given device).

This is illustrated in the figure below.

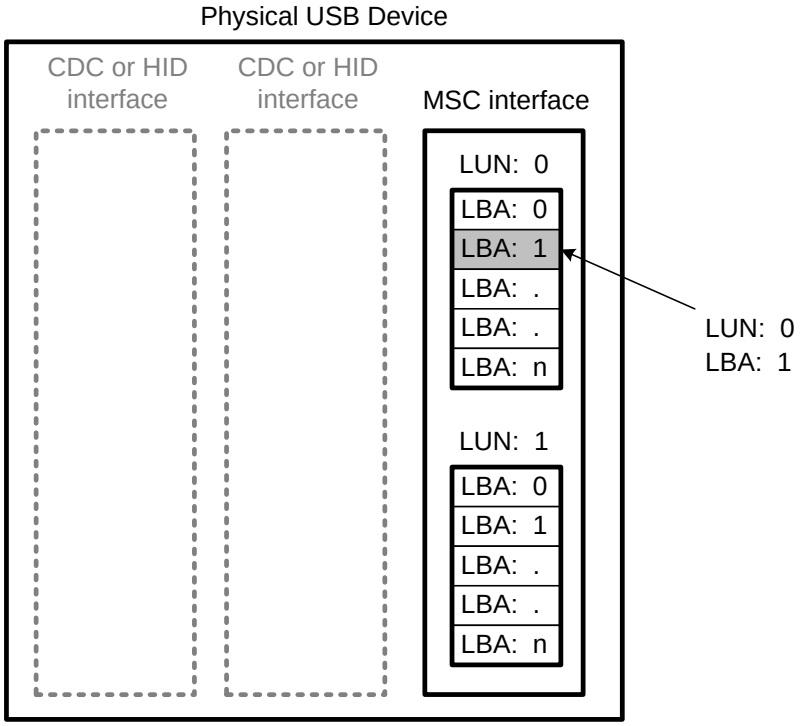


Figure 15. MSC Interface LUN/LBA Addressing

The API supports multiple LUNs, containing any 32-bit number and size of logical blocks. The number of LUNs can be selected within the Descriptor Tool. Since the MSC interface only prepares a substrate upon which the host can mount volumes, each LUN's volume must still 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 access the volume when it receives a SCSI read or write command, it passes the SCSI command's LUN, initial LBA, and number of requested blocks 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).

8.4 MSC Function Call / Event Summary

A table summarizing the MSC API function calls is shown below. For complete documentation, see the API call reference within the USB Developers Package.

Table 15. 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.

Table 16. MSC Event Handler Summary

`BYTE USBMCS_handleBufferEvent()`	Indicates that the API needs for the application to process a buffer.

For reference details on these functions, including description, parameters, and calling conditions, please see the API Functional Reference within the USB Developers Package. The functions are further described below.

8.5 Components of an MSP430-Based MSC Application

The table below shows the responsibilities of the application to handle MSC request. Performing all these duties allows the volume to mount on the host, and be maintained there until removal.

Table 17. Actions the Application Must Take

When	Action	How
Initialization	Define the LUN structure and characteristics	*Define a* `USBMSC_config` *structure*
	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 is used by the API to describe any buffer operations it requests of the application.	`USBMSC_fetchInfoStruct()`
	Inform the API about the medium (if any) initially present for each LUN	`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.

8.5.1 Defining the Interface's LUNs

An application must define its LUNs. The API then uses this definition when responding to SCSI commands.

This is done using the Descriptor Tool. When an MSC interface is added, its view provides a pulldown menu to select the number of LUNs, and the characteristics of each LUN.

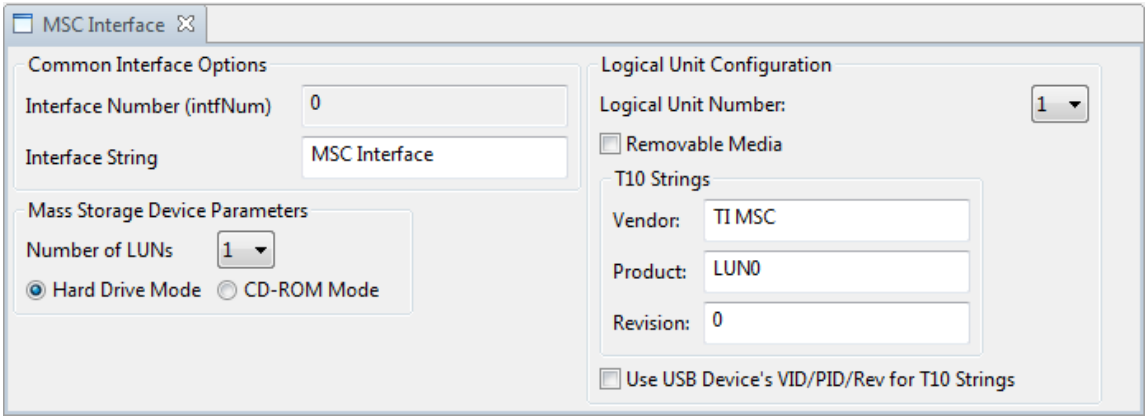


Figure 16. Selecting LUN Attributes

Use the Descriptor Tool's instructions to set these fields. When the output has been generated, it contains a structure named `USBMSC_config`, which reflects the characteristics of the LUNs. The information in this structure is as shown below.

Table 18. LUN Information in the *USBMSC_config* Structure

Field	Format	Description
.LUN.number	BYTE	The logical unit number. The first one is 0x00, and they must increment sequentially.
.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. However, since unique T10 values are needed, the Tool gives the option to borrow these values from the USB vendor/product information.

8.5.2 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 at least one block. A block size is defined as discussed in Sec. 8.5.4; for the FAT file system, a block is 512 bytes. The buffer can be larger than one block; if this is done, it should always be a multiple of the block size.

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 (must be equal)

If the address of the Y-buffer is non-null, this enables *double-buffering* to be used by the API, which increases throughput. If null, then *single-buffering* will be used. Double-buffering enables modest speed gains. (See the Release Notes HTML file, for benchmark information.)

The application is allowed to dynamically change the buffer location. 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()`.

8.5.3 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.

8.5.4 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_setup()` or `USB_connect()`). `USB_setup()` sets the MediaPresent and WriteProtected flags to default values so if changes to these values are required, they must be done before calling `USB_connect()`.
2. If the media is removable, it must also inform the API as soon as after 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 19. USBMSC_mediaInfoStr

Type	Name	Description
BYTE	mediaPresent	Indicates that the medium is present (0x81) or not present (0x82).
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 0x81. .

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.

To reset either mediaPresent or writeProtected flags during program execution, USB has to be first disconnected and disabled, the flag changes done and then USB enabled, reset and connected.

8.5.5 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.

8.5.5.1 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.

8.5.5.2 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, atomically 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()`.

8.5.5.3 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
- Any “long” polling period, causing a host timeout to be violated

The more frequently an application 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 a single long period between `USBMSC_poll()` calls could exceed a host timeout period. The timeout periods vary by operating system and situation. On READs/WRITEs, these tend to be 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.

8.5.6 Processing Buffer Events

8.5.6.1 What Are They?

As discussed in Sec. 8.1, 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.

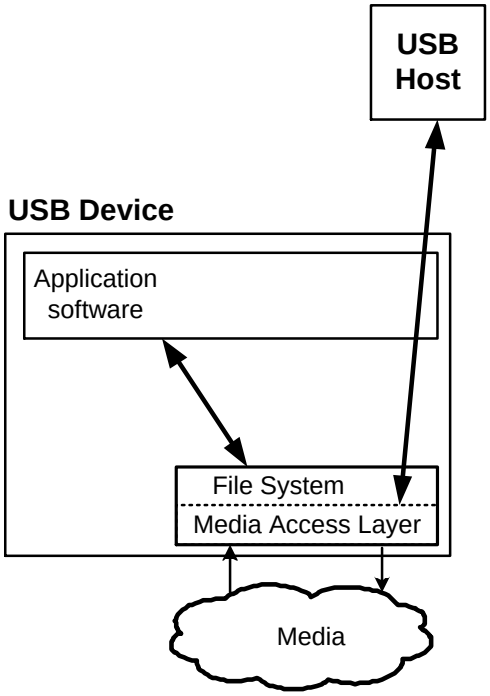


Figure 17. File System Access – Simplified

There is a wide variety of file system middleware on the market, with various specialties. Some are optimized for code size, while some have advanced features. Since users may have different preferences, a file system has not been integrated within 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. (An MSP430 port of the open-source file system software "FatFs" is included in the examples.)

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.

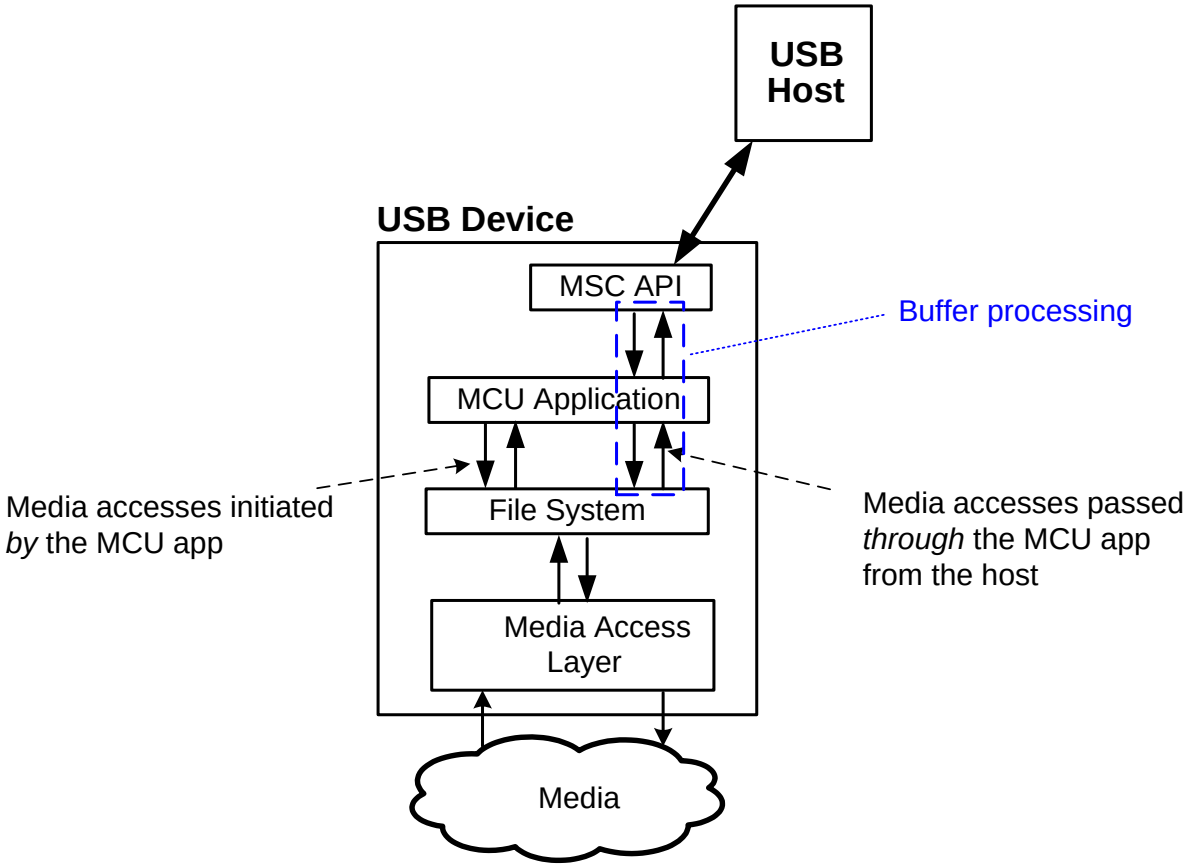


Figure 18. File System Access – Actual

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.

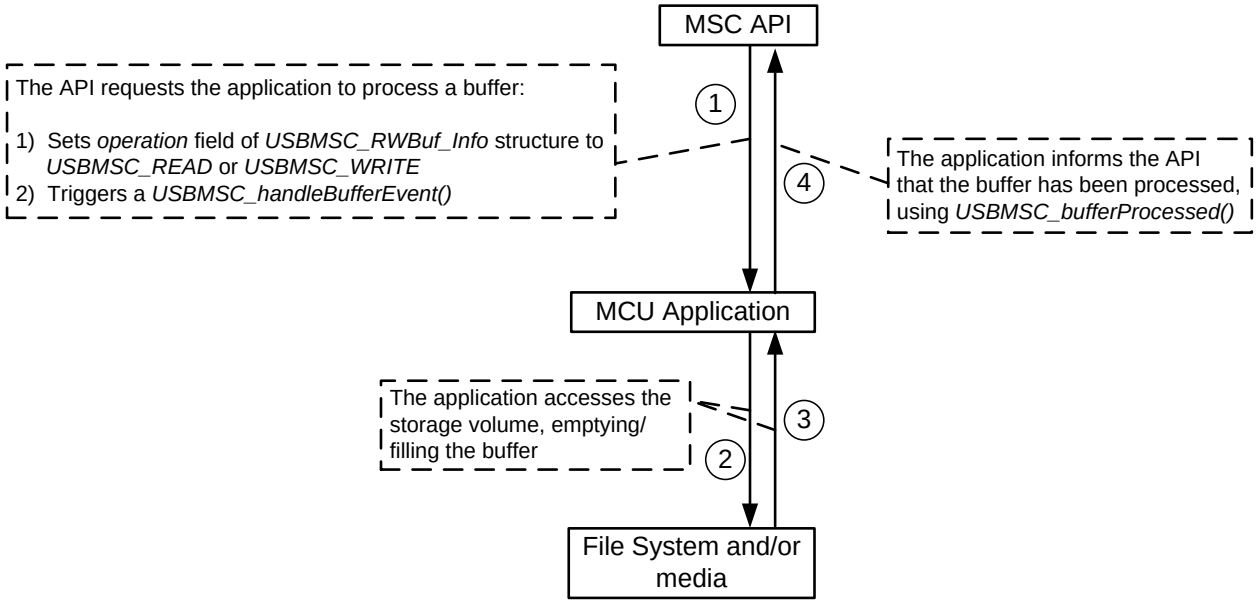


Figure 19. Buffer Processing

Buffer processing is further detailed below.

8.5.6.2 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 20. 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().*

8.5.6.3 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. 8.5.5 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 21. 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.

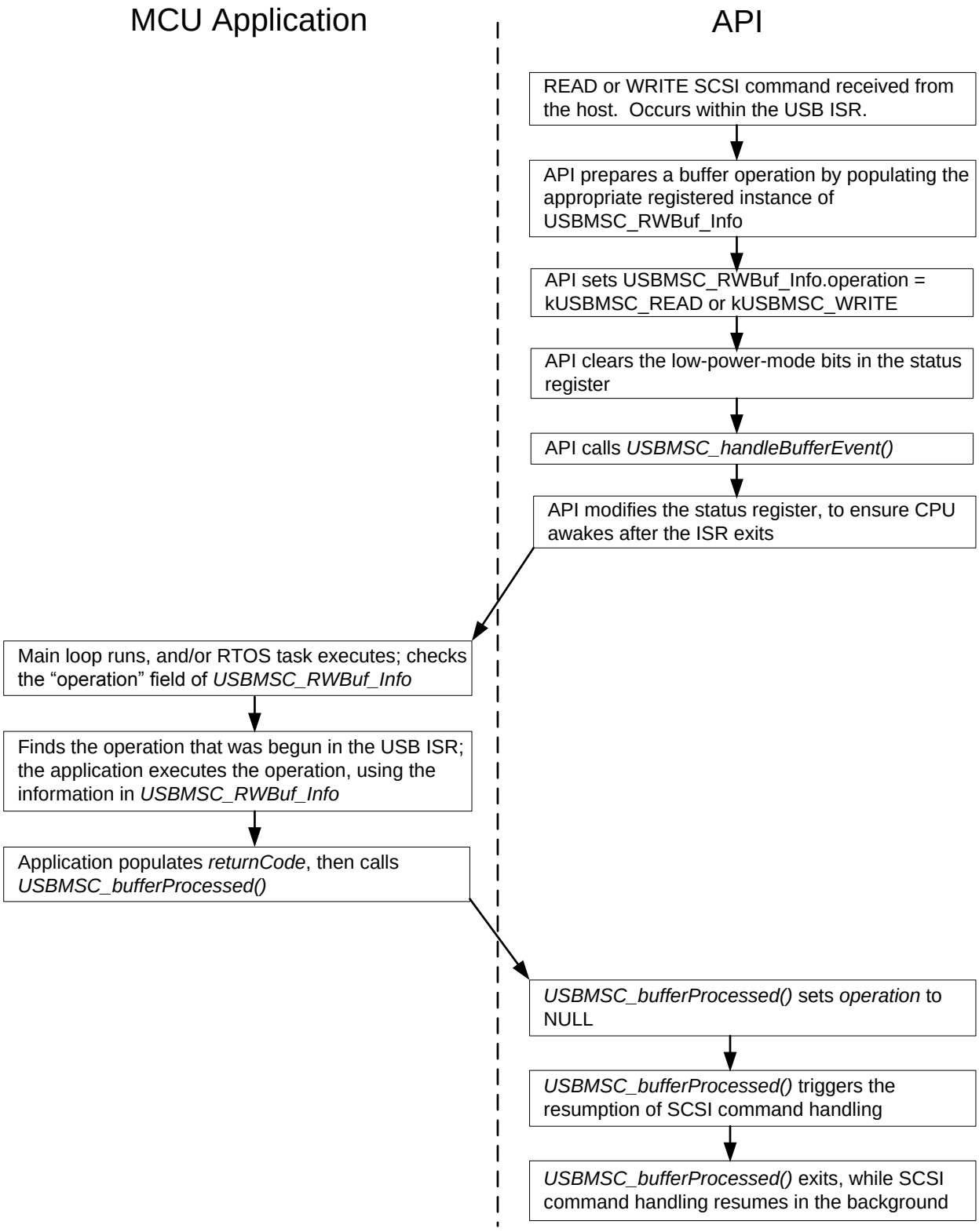


Figure 20. Buffer Lifecycle

8.6 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 over USB.

9 Traditional HID Interfaces

This section applies to traditional HID interfaces, including mice, keyboards, and HID interfaces using custom report formats. This chapter does not apply to datapipe interfaces, such as CDC HID-Datapipe; these are discussed in Sec. 7.

9.1 Introduction

The table below summarizes the different HID types provided within the API. The API pre-defines three sub-types of HID: datapipe, mouse, and keyboard. Any other HID interface is considered 'custom'.

Table 22. HID Interface Types, Within the MSP430 USB API

Functionality	Description	Report Format	How to Create	Function Call Set to Use
Datapipe	UART-like PC/device communication.	A minimal format for simple data transfer	Select 'Datapipe' as the HID interface type in the Descriptor Tool	HID-Datapipe
Mouse	Mouse functionality	Standard mouse report format	Select 'Mouse' as the HID interface type in the Descriptor Tool	HID-Traditional
Keyboard	Keyboard functionality	Standard keyboard report format	Select 'Keyboard' as the HID interface type in the Descriptor Tool	
Custom	Any other HID device/interface	User-customized	Select 'Custom' for this interface type, in the Descriptor Tool; then follow the instructions in this chapter	

HID-Datapipe interfaces use a specific report format, and they use the datapipe function calls. Any HID interface not using both of these are considered HID-Traditional. HID-Datapipe interfaces are discussed in Sec. 7, and not in this chapter.

All HID-Traditional interfaces use the HID-Traditional function calls. Traditional devices are further distinguished by their report formats into mice, keyboard, or custom HID interfaces. The application must also interact with each type differently.

Except for 'custom' HID interfaces, the Descriptor Tool automates much of the interface's creation, as will be described later. The USB Developers Package also has examples of various HID interface types.

HID interfaces types differ in how host-side communication is performed, as shown below.

Table 23. HID Interface Types, Host Considerations

Functionality	Host Considerations
Datapipe	Requires an application to be written, which recognizes the HID-Datapipe report and protocol. TI provides the *Java HID Demo App* for this purpose.
Mouse	The host operating system interacts with the device, rather than a host application.
Keyboard	
Custom	Requires an application to be written. Although the *Java HID Demo App* is written for HID-Datapipe, it can easily be leveraged for custom reports as well.

9.2 Creating a Mouse or Keyboard

Creating a mouse or keyboard with the MSP430 USB API is very simple. To get started, examples are provided in the USB Developers Package: example #H7 implements a single-interface HID mouse, while example #H8 implements a single-interface HID keyboard. These examples can be built and modified for experimentation purposes.

For more serious development, use the Descriptor Tool to create the interfaces. This may especially be desirable if adding a mouse or keyboard in composite with other interfaces.

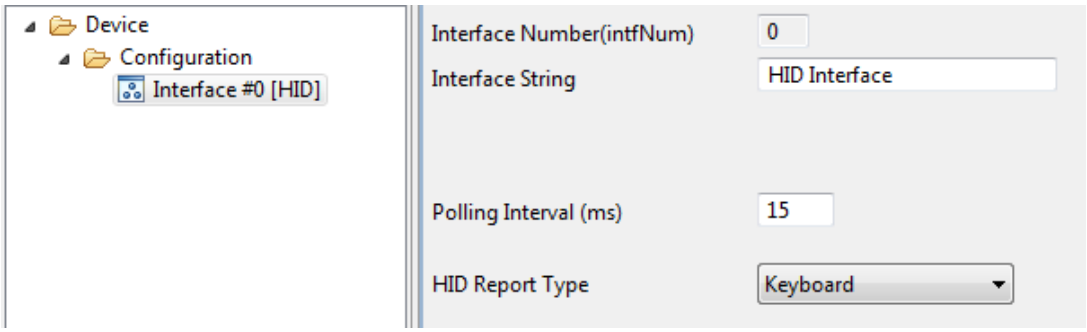


Figure 21. Using the Descriptor Tool to Create a Mouse or Keyboard HID Interface

Using the Tool allows all the custom information to be easily modified. If the mouse/keyboard is to be part of a composite USB device alongside other interfaces, the Tool greatly simplies the process of generating the needed descriptors.

After using the Tool, application code is needed to interact with the newly-created interfaces. The application code from the examples can be borrowed for this purpose. The polling frequency may need to be adjusted within the Tool, according to the application. (This is the rate at which the host will poll for reports from the HID interface.)

The standard mouse report descriptor, and resulting report format, are shown below.

```
BYTE const report_desc_HIDx[] =
{
    0x05, 0x01,         // Usage Pg (Generic Desktop)
    0x09, 0x02,         // Usage (Mouse)
    0xA1, 0x01,         // Collection: (Application)
    0x09, 0x01,                 // Usage Page (Vendor Defined)
    0xA1, 0x00,         // Collection (Linked)
    0x05, 0x09,         // Usage (Button)
    0x19, 0x01,         // Usage Min (#)
    0x29, 0x05,         // Usage Max (#)
    0x15, 0x00,         // Log Min (0)
    0x25, 0x01,                 // Usage Maximum
    0x95, 0x05,         // Report count (5)
    0x75, 0x01,         // Report Size (1)
    0x81, 0x02,         // Input (Data,Var,Abs)
    0x95, 0x01,         // Report Count (1)
    0x75, 0x03,         // Report Size (3)
    0x81, 0x01,         // Input: (Constant)
    0x05, 0x01,         // Usage Pg (Generic Desktop)
    0x09, 0x30,         // Usage (X)
    0x09, 0x31,         // Usage (Y)
    0x09, 0x38,         // Usage (Wheel)
    0x15, 0x81,         // Log Min (-127)
    0x25, 0x7F,         // Log Max (127)
    0x75, 0x08,         // Report Size
    0x95, 0x03,         // Report Count (3)
    0x81, 0x06,         // Input: (Data, Variable, Relative)
    0xC0,               // End Collection
    0xC0                // End Collection
};
```

Figure 22. Mouse Report Descriptor

The mouse report descriptor describes the report shown in the table below.

Table 24. Mouse Report Format

Field	Size	Description
IN report (into the host)		
Buttons	1 byte	3 button bits and padding
dX	1 byte	X position
dY	1 byte	Y position
dZ	1 byte	Wheel position.

The standard report descriptor for keyboards, and the resulting report format, are shown below.

```
BYTE const report_desc_HIDx[]=
{
    0x05, 0x01,   // Usage Page (Generic Desktop)
    0x09, 0x06,   // Usage (Keyboard)
    0xA1, 0x01,   // Collection (Application)
    0x05, 0x07,   // Usage Page (Key Codes)
    0x19, 0xE0,   // Usage Minimum (224)
    0x29, 0xE7,   // Usage Maximum (231)
    0x15, 0x00,   // Logical Minimum (0)
    0x25, 0x01,   // Logical Maximum (1)
    0x75, 0x01,   // Report Size (1)
    0x95, 0x08,   // Report Count (8)
    0x81, 0x02,   // Input (Data, Variable, Absolute) -- Modifier byte
    0x95, 0x01,   // Report Count (1)
    0x75, 0x08,   // Report Size (8)
    0x81, 0x03,   // (81 01) Input (Constant) -- Reserved byte
    0x95, 0x05,   // Report Count (5)
    0x75, 0x01,   // Report Size (1)
    0x05, 0x08,   // Usage Page (Page# for LEDs)
    0x19, 0x01,   // Usage Minimum (1)
    0x29, 0x05,   // Usage Maximum (5)
    0x91, 0x02,   // Output (Data, Variable, Absolute) -- LED report
    0x95, 0x01,   // Report Count (1)
    0x75, 0x03,   // Report Size (3)
    0x91, 0x03,   // (91 03) Output (Constant) -- LED report padding
    0x95, 0x06,           // Report Count (6)
    0x75, 0x08,           // Report Size (8)
    0x15, 0x00,           // Logical Minimum (0)
    0x25, 0x66,           // Logical Maximum(102)  // was 0x65
    0x05, 0x07,           // Usage Page (Key Codes)
    0x19, 0x00,           // Usage Minimum (0)
    0x29, 0x66,           // Usage Maximum (102) // was 0x65
    0x81, 0x00,           // Input (Data, Array) -- Key arrays (6 bytes)
    0xC0                  // End Collection
};
```

Figure 23. Keyboard Report Descriptor

Table 25. Keyboard Report Format

Field	Size	Description
IN report (into the host)		
Modifier	1 byte	Modifier keys (SHIFT, CTRL etc)
Reserved	1 byte	reserved
Key Arrays	6 bytes	Non-modifier key
OUT report (out of the host)		
LED	1 byte	LED report and padding

9.3 Creating a Custom HID Interface

The API and Descriptor Tool pre-define mice, keyboards, and datapipe report formats. Any other report format results in a “custom” HID interface.

The developer must generate the report descriptor that describes this custom report. General information about doing so is given below.

Developing a custom HID device is more complicated than some other interfaces, and so the developer should have a clear idea why it’s being chosen. There are generally two such situations:

- PC peripherals other than mice/keyboards
- Maintaining backward-compatibility with an existing host application that already uses a custom report format

The HID class was originally created for PC peripherals. The basic unit of data exchange was called a *report,* which could be of varying formats. A USB device communicates its report format using what is essentially a complex tag/scripting language. This provided flexibility, allowing one device class to be used with a wide variety of PC peripherals. If creating a peripheral, where the host OS itself handles the interaction with the device, and if the functionality isn’t mouse/keyboard, then a custom report is required, and therefore a custom HID interface.

Although initially intended for OS-driven peripherals, HID has also been used in general-purpose applications where a host application program interacts with the device. HID-Datapipe is an example of this. However, if the developer is trying to maintain compatibility with an existing host application that uses a particular report format, then HID-Datapipe won’t work with it; HID-Datapipe has its own report format. As such, a custom HID interface on the MSP430 is required, as well as a report descriptor describing the existing report format.

To create a custom HID interface, the recommended approach is:

1. Use the Descriptor Tool to generate a HID interface . For the HID report type, select ‘Custom’.

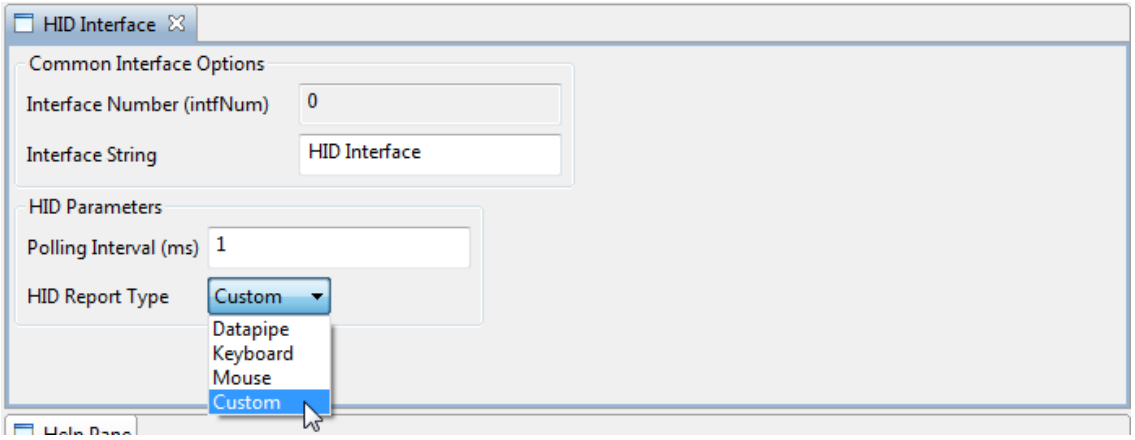


Figure 24. Creating a Custom HID Device

2. A pop-up text box allows the custom report descriptor to be input. (See Sec. 9.4 for how to generate this.)

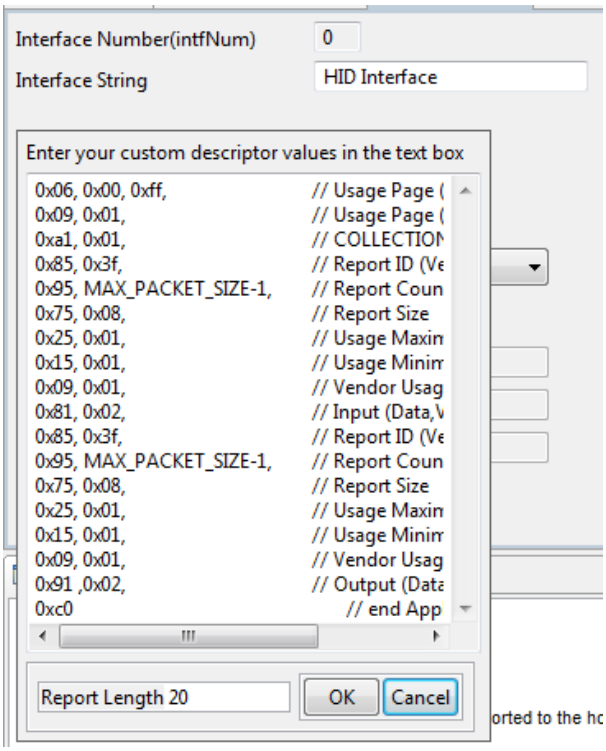


Figure 25. Entering a Custom HID Report Descriptor into the Descriptor Tool

3. Input the report's data length at the bottom of this popup window, in bytes. Note that this is not the length of the report descriptor, but the length of the reports that will be transferred in normal operation.
4. Be sure to choose the correct polling interval (in the "interface view") required for your application.
5. In the application, use the HID-Traditional function calls to send and receive HID reports

See Sec. 5 for more information on using the Descriptor Tool.

9.4 Generating a Custom HID Report Descriptor

If generating a custom HID interface, the developer must provide a report descriptor.

Report descriptors may come from these sources:

- If the function is a PC peripheral, in which the host OS handles the interaction, then the OS documentation may specify a required report format.
- Examples may be found in the public domain
- The USB-IF's HID Descriptor Tool

The first two sources tend to provide intact report descriptors that can be pasted directly into the MSP430 USB Descriptor Tool.

If the developer is creating a report truly from scratch, the HID Descriptor Tool can be used. This is a tool provided by the USB Implementers Forum (USB-IF) to aid the creation of HID report descriptors. Note that despite the similar name, the functionality of the USB-IF's tool is very different than the MSP430 HID Descriptor Tool.

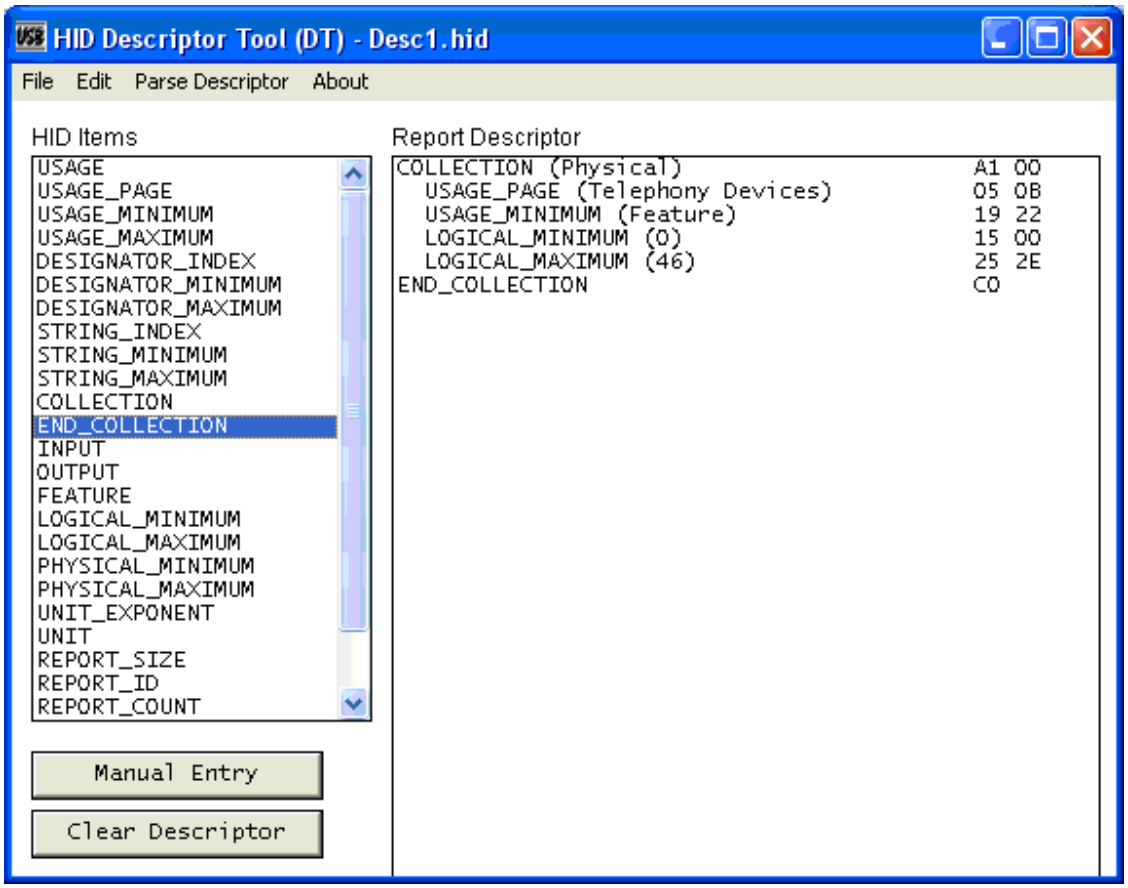


Figure 26. The USB-IF's HID Descriptor Tool

As seen in the figure above, even though the HID Descriptor Tool is helpful with formatting and syntax, it still requires an understanding of the complex language used to define HID reports. This knowledge is outside the scope of this Programmer's Guide; the developer motivated to learn this language can reference materials from the USB-IF and other sources.

Once the report format is created, the output from the HID Descriptor Tool is a series of hex values that can be pasted directly into the MSP430 USB Descriptor Tool's pop-up text box , as described in the previous section.

9.5 Writing Application Code for Traditional HID Devices

Traditional HID interfaces (any HID interface other than HID-Datapipe) use the HID-Traditional function calls.

Table 26. HID-Traditional Function Call Summary

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

Table 27. HID-Traditional Event Handler Summary

`BYTE USBHID_handleSendCompleted()`	Interrupt for the input endpoint associated with this data interface	Indicates that an INPUT report has been fetched by the host, for HID interface *intfNum*
`BYTE USBHID_handleReceiveCompleted()`	Interrupt for the output endpoint associated with this data interface	Indicates that an OUTPUT report has been received from the host, for HID interface *intfNum*

For reference details on these functions, including description, parameters, and calling conditions, please see the API Functional Reference within the USB Developers Package. The functions are further described below.

The application must allocate and populate a report buffer (up to 64 bytes), and then pass it into the API using `USBHID_sendReport()`. This buffer contains, byte for byte, the report that will be given to the host the next time the host polls this interface for a report. It is up to the application to format these 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 is best to call it in response to a `USBHID_handleDataReceived()` event.

In either case, the report length is defined by `USBHID_REPORT_LENGTH`, in *descriptors.h* .

If the HID interface defined a custom report descriptor, this descriptor might have defined multiple reports, each with a unique “report ID” field. The application can therefore send/receive multiple reports by setting the correct report ID value prior to calling `USBHID_sendReport()`, and similarly can separate reports once received with `USBHID_receiveReport()` by reading the report ID field.

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

10 Event Handling

The USB interrupt service routine (ISR) is a resource owned by the API, and is not intended to be modified. Rather, the API generates *events* the application sometimes need to handle. Most of the defined events are called from within the USB ISR, in response to various USB interrupts.

Developers with a hardware background may think of event handlers as similar to ISRs. Using the function `USB_setEnabledEvents()` function can be thought of similarly to setting/clearing interrupt enable bits. Developers with a software background may think of event handlers as pre-defined callback functions.

The handler functions are located in `USB_eventhandling.c`. This file is considered to be an application-owned resource; the software designer has freedom to modify each handler's content (but not the function's declared parameters).

10.1 The Relationship between Interrupts and Events

The vast majority of event handlers are called out of the USB ISR. As a result, general interrupts are disabled during the event handlers, and 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 avoids delays in handling other interrupts. If this isn't possible, then another option is to set a flag within the event handler and handle the function in main(); that is, no longer in an interrupt-handling context. (If main() makes use of LPM0, it may be necessary for the USB ISR to keep the MSP430 awake after the ISR returns; see the next section for more information.)

Another recommendation is to not enable general interrupts (GIE) within the event handler. The MSP430 disables general interrupts during event handlers, to avoid the potential for a stack overflow. Enabling interrupts within an event handler would incur this risk.

10.2 Waking from Event Handlers

MSP430 coding frequently makes 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);
```

This intrinsic reaches into the stack and modifies the status register bits indicating what power mode the MSP430 should return to after the ISR, forcing them to an active CPU state. Therefore, if the MSP430 was in a low-power mode when the interrupt occurred, then the CPU will resume execution when the ISR returns, starting from the point at which 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 after return. The developer may wish to reduce interrupt latency by simply setting a flag in the event handler, and do the actual handling in main(), triggered by the flag. Event handlers can accomplish this by returning TRUE when exiting. This forces the CPU to active mode. Returning a value of FALSE causes the CPU to return to whatever power state was active when the ISR was triggered.

10.3 Calling API Functions from Event Handlers

Generally speaking, it isn't recommended to call datapipe sending functions from within an event handler. If the interface should happen to be busy at the point of sending, then it won't become available until after the event handler (and USB ISR) returns. Performing these functions in a non-ISR context allows the USB interface to finish any open operations before starting a new one. 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 above.

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

10.4 Enabling Events

The event handlers only run in response to an event if their respective event is *enabled.* By default, they are all disabled. However, if the function `USB_setup()` is used at initialization, it enables all available events. Further enabling/disabling can be done with `USB_setEnabledEvents()`.

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.

10.5 Event Handler Functions

A summary of the event handler functions is shown in the table below. For complete information, see the Doxygen reference within the USB Developers Package. Also see discussion of various events elsewhere within this Programmer's Guide.

10.6 Event Handler Summary

The event handlers are summarized below.

For reference details on these functions, including description, parameters, and calling conditions, please see the API Functional Reference within the USB Developers Package.

Table 28. Summary of Event Handlers

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

Table 29. Summary of Event Handlers (continued)

`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 endpoints associated with this interface	Indicates that the API needs the application to process a buffer.

11 Practical Matters: Writing USB Programs with the API

11.1 Power Management

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.

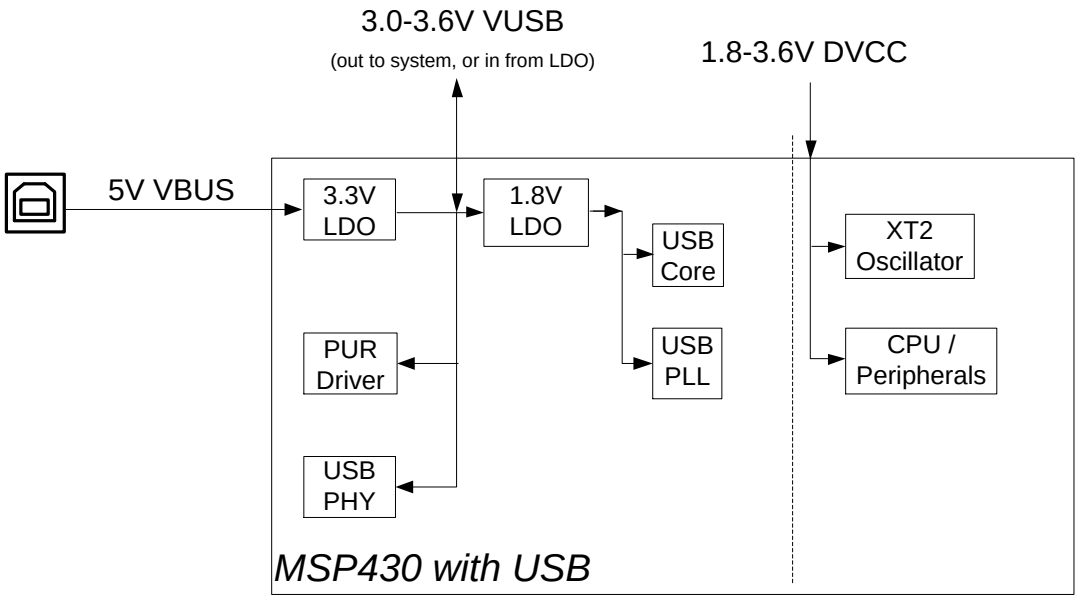


Figure 27. MSP430 Power Flow, with Respect to USB

Note: This is not a schematic. Please see electrical reference circuits for guidance on the hardware.

The developer has some options with respect to the VUSB rail. Using the internal LDO's, VBUS can source the USB module. The LDO has sufficient output capability to drive other components in the system, or to the DVCC pin; see the device datasheet for maximum output current.

11.1.1 Directing Power into VUSB from an External Source

It's possible to drive power into the VUSB rail from an external source, rather than using the integrated 3.3V LDO. The reason this might be desirable is that the quiescent current on the integrated 3.3V LDO is in area of 100uA. Even though host power is usually available to power this, in some cases the host's supply itself might be limited. If an external LDO is present nearby, it can be directed into VUSB.

If this is done, the 3.3V LDO should be disabled. An option is available in the Descriptor Tool to do this automatically, shown below.

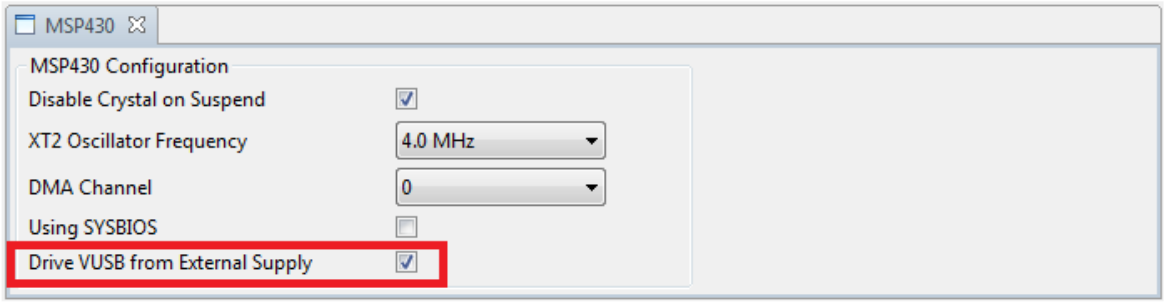


Figure 28. Descriptor Tool Selection for External VBUS Source

Please be aware that the on-chip USB BSL is not designed to operate this way; it always enables the integrated LDO. If this operation is desired, a small software change can be made to the BSL source (available as app note *slaa450*), and the modified BSL can be re-programmed into the MSP430.

If this option is chosen, it's still important to connect 5V VBUS to the USB connector, because this is the way a USB device is able to tell that a host is present.

11.1.2 The API's Effect on Power Settings

The only USB API call that affects the MSP430's power settings is `USB_setup()`. (`USB_init()` does as well, but this function has been deprecated in v4.0.) `USB_setup()` enables the 3.3V and 1.8V USB LDOs. The USB API does not affect the MSP430F5xx's VCORE setting.

11.1.3 VCORE Setting

The MSP430F5xx's Power Management Module (PMM) includes an LDO (separate from the ones in the USB module) that sources a nominal 1.8V to the digital core. There are actually four settings for VCORE: 0-3. Usually the choice of VCORE setting is determined by the MCLK speed.

When the USB PLL is enabled, however, the minimum required setting is "2", no matter the speed of MCLK. The USB PLL is enabled with `USB_enable()` and disabled with `USB_disable`. The PLL is also automatically disabled during USB suspend. If VCORE is set to "0" or "1" when the PLL is enabled, errors will result.

The VCORE setting is controlled by the PMMCOREV field in the PMMCTL0 register. It's recommended to configure the PMM using the driverlib call `PMM_setVCore()`. A copy of driverlib is included with all the examples and empty USB project.

11.1.4 Managing VBUS Power During USB Suspend

When the host suspends a USB device, the current being drawn from the host via VBUS must be reduced. It must do this within 7ms of the suspend event, which the application sees as a `USB_handleSuspendEvent()`. The application may need to shut down circuitry to avoid violating this USB requirement.

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.

If a crystal is used on XT2 to drive the USB PLL (typically the case in most applications), then 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.)

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 `USB_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.

11.1.5 Selective Suspend

As discussed in Sec. 6.8, from a USB device's standpoint, there is no difference between a whole-bus USB suspend, and selective suspend. The USB device only knows that it's been suspended. The host determines whether to selectively suspend a device based on a set of rules determining whether the device has been inactive for a sufficient period of time.

The only way in which the USB device can prevent entry into selective suspend is if the nature of communication between the device and host is such that the operating system decides that the device is not idle. For example, the most may have certain timeout periods that must expire, and perhaps the USB device is communicating too frequently. It may be beneficial to consult with the operating system's technical documentation to determine whether this is the case.

There are sometimes settings on the host operating system that are required to enable selective suspend. Some of these settings might be required on a per-device basis. HID on Windows sometimes requires a special INF file that enables selective suspend.

11.1.6 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 in a manner similar to any MSP430 application. The only restriction is that while the USB PLL is enabled, the power mode should not be "less" than LPM0 (in other words, do not use LPM3/4/5). The USB PLL is enabled with `USB_enable()` and disabled with `USB_disable()`.

The PLL is also automatically disabled during USB suspend. During USB suspend, the CPU 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 and can happen at any time, 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.

11.2 Clock Management

11.2.1 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. The PLL is sourced directly from XT2.

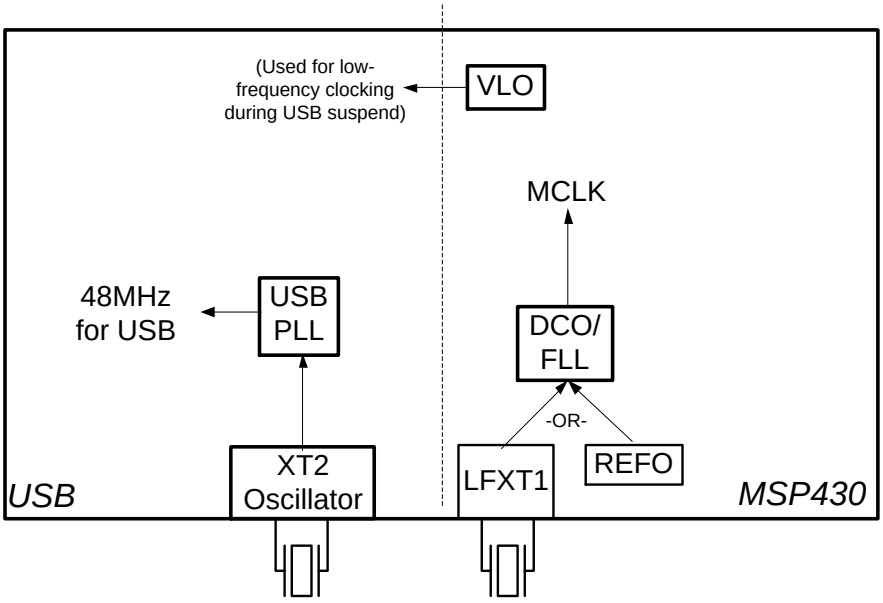


Figure 29. Recommended Clocking in a USB-Equipped MSP430

XT2 is considered an API-owned resource. The API enables and disables it as needed. Some configurability of this behavior is provided in the Descriptor Tool. The Tool also sets certain values the API needs to properly manage XT2 and the PLL.

During USB suspend, the API disables the PLL, and optionally XT2. During this time, the USB module begins receiving clocking from the always-on VLO (very low-frequency, low-power oscillator). This low-power oscillator is sufficient for the module's functions during suspend.

11.2.2 MCLK Requirements

USB operation can take place over a wide range of MCLK values. Most testing has taken place with MCLK at least 4MHz, but this may not be the practical lower limit.

The application developer has complete freedom to set MCLK speed. The driverlib library accompanying the examples and empty USB project can be used to set the system clocks.

It is not supported to use REFO, LFXT1, or the VLO for MCLK.

11.2.3 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 Descriptor Tool can place the XT2 in Bypass mode via the value USB_XT2_BYPASS_MODE.

The PLL reference must adhere to certain frequency, precision, and jitter requirements. (See the device datasheet.) Crystals driven by the XT2 oscillator easily meet these requirements. Some ceramic resonators do as well.

But if bypass mode is used -- meaning a clean clock is driven into XT2's input pin -- the designer must consider these requirements. The tolerance requirement is provided by the USB specification: ±2500ppm. The MSP430's DCO/FLL output does not meet the PLL's 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 or compliant ceramic resonator 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

11.2.4 API Delay Loops

The API employs CPU-driven delay loops in a couple places to implement delays required by the USB module. Such delays must compensate for varying MCLK speeds. Prior to v4.0 of the API, it obtained this information from the Descriptor Tool, via the value USB_MCLK_FREQ. Since this was a compile-time value, it did not adapt to any changes in MCLK over time.

As of v4.0, an internal function is employed that determines the MCLK speed from the UCS registers. The delay is adjusted according to the perceived clock speed. As a result, there's no longer a need to input this value via the Descriptor Tool, and it always adjusts to whatever MCLK is set to at run-time.

11.2.5 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.

11.2.6 Using XT2 for Non-USB Functions

Although XT2 is considered an API-owned resource, it's possible for it 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.

It should be kept in mind that crystal precision generally has no advantage for MCLK; the CPU will not execute any differently with crystal precision. Crystal precision may be useful for certain analog/digital conversion functions.

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.

11.3 "Bus Errors"

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 is 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.

This error should not occur in normal operation, because the API knows not to access USB RAM when the PLL is inactive. If it does happen, it might be a sign that the PLL (or XT2) failed unexpectedly.

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. Applications should contain the code to handle this.

11.4 Use of DMA

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

11.5 Using an RTOS

See Sec. 13 for a discussion on using the USB API with real-time operating systems (RTOSes).

11.6 System Interrupts

USB operation involves a large number of interrupts. It's important that the developer consider the impact of the non-USB operation on the USB interrupt handling, and vice versa.

11.6.1 Ensuring the MSP430 Services USB Interrupts

In some cases the host enforces timeout periods if the device doesn't respond in time. Usually the critical timeouts are services at a hardware level, and in the majority of cases handled by software, the device is able to "NAK" (no-acknowledge) the host, essentially asking it to wait until the device is ready. However, in some cases and with enough delay, the application software may be capable of causing the host to timeout.

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 30. 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.

11.6.2 USB ISR Latency

On the MSP430, new interrupts are disabled while an interrupt service routine is handling an interrupt. This makes latency important – the time required for the USB ISR to complete. For most sources of USB interrupts, the latency is quite small. (See the Release Notes HTML file in the USB Developers Package for specific metrics.)

An exception is when the host resumes the USB device out of USB suspend. At these times, `USB_enable()` is called. This function starts the XT2 oscillator, if it wasn't already running. This can take several hundred microseconds. The PLL must also then be locked, and the datasheet indicates this can take up to 2ms. (See the device datasheet for up-to-date parametrics.)

If this is unacceptable in your application, it's possible to divide the USB resume functionality into three phases, most of which does not take place within an interrupt context. The tradeoff is that the application must take responsibility to ensure that the XT2 oscillator has stabilized, and that the PLL delay period has expired. Example #G1 in the USB examples shows how to do this; the example is also explained in detail in the Examples Guide. The divided functions are described in Sec. 6.1

11.7 USB Design Considerations

11.7.1 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 return values that help in this effort.

11.7.2 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_PHYS_DISCONNECTED`, `ST_ENUM_ACTIVE`, and `ST_ENUM_SUSPENDED`. The state can change at any time.

For some 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 under these conditions.

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.

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.

Such devices are referred to below as *state-dependent.*

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 for other devices, USB sending and receiving is incidental to the main application, and the application is relatively unaffected if data sending fails due to a detached bus, or if data is never received.

These devices are referred to below as *state-independent.*

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

Table 31. 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 mass storage; 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
Data sensor/logger (example)	Logs data into internal MSP430 flash	Continues to log data, but also sends log entries over USB	Logs data in internal MSP430 flash

Therefore, the designer should consider how (or if) the USB state will impact functionality.

11.8 State-Dependent Functionality: Main Loop Framework

One option for state-dependent devices is a main loop construct like the one shown below.

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

    while(1)
    {
        switch(USB_connectionState())
        {
          case ST_ENUM_ACTIVE:
             __bis_SR_register(LPM0_bits + GIE);  // Enter LPM0
             function_as_USB_active();
             break;
          case ST_PHYS_DISCONNECTED:
          case ST_ENUM_SUSPENDED:
          case ST_PHYS_CONNECTED_NOENUM_SUSP:
             __bis_SR_register(LPM3_bits + GIE);  // Enter LPM3
             function_as_USB_inactive();
             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 alternate main loops. As such, USB state becomes a central part of managing software flow.

In this particular example, software behaves the same whether the device is attached to a host but suspended, not attached to a host, or in `ST_PHYS_CONNECTED_NOENUM_SUSP`, which generally means a powered hub without a host present, or an unresponsive host. In those cases, the main loop functionality is determined by `function_as_USB_inactive()`. If the device is not attached and then becomes so, `USB_handleVbusOnEvent()` can be configured to initiate a connection, changing the state to `ST_ENUM_ACTIVE`. (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.)

When enumeration completes, the main loop waits for an interrupt to begin executing `function_as_bus_active()`.

Notice that the four events shown are enabled and set to return TRUE. These allow the main loop be "refreshed" to reflect the new state that result after these events. 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.

Most of the examples and the empty USB project reflect a similar approach to the one shown above.

11.9 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_SystemAndTimer();
    USB_setup();

  while(1)
  {
    __bis_SR_register(LPM0_bits + GIE);           // LPM3 if possible, but must
    process_data();                                // Do what we woke up to do

    prepare_buffer(buffer,size);
    USBCDC_sendData(buffer,size,0);  // Return value is ignored, because success or
                                     // failure doesn't affect program flow

  }  // 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.

The example enters LPM0 in all cases, even though it could be entering LPM3 if USB wasn't active. This was done for simplicity, but in an actual power-sensitive application, LPM3 may be desirable.

This example shows the use of events to manage power. If USB is present, the system relies on VBUS for power. Otherwise it uses a battery. Using VBUS during USB may be essentail for a battery-powered USB system, because staying active on USB requires considerable power

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.

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.

11.10 Tips for Sending Data over Datapipe Interfaces

11.10.1 Conditions to Consider When Sending Data

When using `USBCDC_sendData()` and `USBHID_sendData()` to send data over datapipe interfaces, two things need to be considered:

- *Background Processing* (or "asynchronous" operation). These functions only begin a send operation and then return. The actual send operation can continue to execute after `USBxxx_sendData()` returns. Software must anticipate that a previous operation may still be in progress, before making another call to `USBxxx_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.

To account for these, TI provides example construct functions that take these concerns into account. The developer is free to edit them, or use them as-is.

Table 32. Construct Functions for Datapipe Sending

Construct Function	Description
`cdcSendDataInBackground()`	Calls `USBCDC_sendData()` asynchronously, checking first for an open interface and exiting if the bus isn't available
`cdcSendDataWaitTilDone()`	Calls `USBHID_sendData()` synchronously; does not return until the data has all been sent, or until it's been determined the host is not available.
`hidSendDataInBackground()`	Same as `cdcSendDataInBackground()`, except using an HID-Datapipe interface.
`hidSendDataWaitTilDone()`	Same as `cdcSendDataWaitTileDone()`, except using an HID-Datapipe interface.

Equivalent construct functions are provided for CDC and HID-Datapipe, reflecting the symmetry between these two interfaces. The prefixes are `cdc` or `hid`, respectively. Since the text in this section applies equally to both types, they're frequently shown in this section with the prefix `xxx` -- for example, "`xxxSendDataInBackground()`".

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

11.10.2 Background Processing

As discussed in Sec. 7.4.2, background send/receive operations increase efficiency, and they avoid blocking MSP430 software execution if the host or bus are slow. 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.

There are three general approaches to handling background processing, shown below.

Table 33. Approaches to Background Processing

Approach	How	Tradeoff
Eliminate it	After calling USBxxx_sendData(), poll the interface's status until it's no longer busy. Or simply call `xxxSendDataWaitTilDone()`.	Simplest to code, but blocks execution until the transfer is complete.
Fully take advantage of it	Arrange software to trigger new send operations in response to the last one ending	More coding effort, but is the most efficient and least blocking.
Mostly take advantage of it	Before calling `USBxxx_sendData()`, poll the interface's status until it's no longer busy. Or simply call `xxxSendDataInBackground()`.	Allows most of the sending to take place during other software execution, but could still end up spending time polling the interface if it's still busy handling a previous send operation.

The first and last approaches are simple to implement within a function, and so they are provided with the API in the form of the "construct" functions `xxxSendDataWaitTilDone()` and `xxxSendDataInBackground()`. (The CDC ones are shown as examples.)

```
BYTE cdcSendDataWaitTilDone (BYTE* dataBuf,
    WORD size,
    BYTE intfNum,
    ULONG ulRetries)
{
  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 here, 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 (ulRetries && (sendCounter++ >= ulRetries)) // Incr & try again
      {
          return (1) ;                               // Timed out
      }
    }
    else
    {
       return (0);                                   // If succeeded
    }
}
```

The function `hidSendDataWaitTilDone()` is identical to this, except that it's for HID-Datapipe interfaces.

This function doesn't return until all the data has been transferred (or an error has occurred). 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 can block execution in the event of a slow host or busy bus.

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.

The function exits if the host is unavailable, or if the interface takes too long to finish its previous operation (optional). If they return a zero value, all went well. If the return value is non-zero, it did not go well; the value should be evaluated and handled accordingly.

An example of calling this function twice in succession is shown below.

```
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 30. Calling xxxSendDataWaitTilDone()

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()` guarantees 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 retry value. This retry is an inexact mechanism, but it effectively ensures that executation doesn't stay here forever.

`cdcSendDataInBackground()` is shown below.

```
BYTE cdcSendDataInBackground (BYTE* dataBuf,
    WORD size,
    BYTE intfNum,
    ULONG ulRetries)
{
    ULONG sendCounter = 0;
    WORD bytesSent, bytesReceived;

    while (USBCDC_intfStatus(intfNum,&bytesSent,
                &bytesReceived) & kUSBCDC_waitingForSend)
    {
      if (ulRetries && ((sendCounter++) > ulRetries)){ // 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);
    }
}
```

In this approach, the order of events is reversed. The interface is polled, and if not busy, data is sent using `USBxxx_sendData()` and returns, while the API sends the data in the background. If `xxxSendDataInBackground()` is called again at a later time, there's a good chance the data has been fully transferred. If not, then the polling time is less than it would have been with `xxxSendDataWaitTilDone()`.

The disadvantage relative to `xxxSendDataWaitTilDone()` is that 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. (This can be handled using an X/Y buffer scheme -- see below.)

The function exits if the host is unavailable, or if the interface takes too long to finish its previous operation (optional). If they return a zero value, all went well. If the return value is non-zero, it did not go well; the value should be evaluated and handled accordingly.

Figure 31. Calling *xxxSendDataInBackground()*

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, which is the reason why two are necessary. 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 retry value. Large values are recommended.

A third approach to background processing is true "asynchronous" sending. Each send is triggered by the completion of the previous operation. As such, there's no need to ever poll `USBxxx_intfStatus()`. Sending is triggered by `USBxxx_handleSendCompleted()`.

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, it's a good choice for 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 and/or more natural to implement in RTOS-based implementations that want to completely eliminate the possibility of polling, or if maximum CPU efficiency is required.
- Use caution if mixing approaches. 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.

11.10.3 Anticipating a Lost Bus

The USB connection might be disconnected or suspended at any time; and as Sec. 11.6.2 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.

11.11 Tips on Receiving Data over CDC or HID-Datapipe

When sending data over USB, the application controls:

- the point within application flow where the transfer will be initiated, and
- how many bytes will be sent

But when receiving data, neither of these is certain to the application; the host controls both factors.

This dictates the 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 34. Receive Construct Options

	Size Known Ahead of Time?	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	(Not applicable)	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 , equivalent receive construct functions are provided for both CDC and HID-Datapipe, reflecting the symmetry between these two interfaces. The prefixes are `cdc` or `hid`, respectively. Since the text in this section applies equally to both types, they're frequently shown in this section with the prefix `xxx`-- for example, "`xxxReceiveDataInBuffer()`".

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

11.11.1 cdcReceiveDataInBuffer() and hidReceiveDataInBuffer()

The `xxxReceiveDataInBuffer()` function opens a receive operation only for data that has already been received into the USB endpoint buffers; this operation completes immediately. (Recall that the endpoint buffers contain a single packet of data from the host – see Sec. 7.)

The benefit of this is that the application doesn't need to worry about the future reception of data; it only polls to see what is already present. This method works best when the expected block of data is small in size, but the exact number of bytes is unknown.

The CDC version of the function is shown below.

```
// This call only retrieves data that is already waiting in the USB buffer -
// that is, data that has already been received by the MCU.  It assumes a previous,
// open receive operation (began by a direct call to USBxxx receiveData()) is NOT
// underway on this interface; and no receive operation remains open after this
// call returns.  It doesn't check for kUSBxxx_busNotAvailable, because it doesn't
// matter if it's not.  size is the maximum that is allowed to be received before
// exiting; i.e., it is the size allotted to dataBuf.  Returns the number of bytes
// received.
WORD cdcReceiveDataInBuffer (BYTE* dataBuf, WORD size, BYTE intfNum)
{
    WORD bytesInBuf;
    WORD rxCount = 0;
    BYTE* currentPos = dataBuf;

    while (bytesInBuf = USBCDC_bytesInUSBBuffer(intfNum))
    {
      if ((WORD)(currentPos - dataBuf + bytesInBuf) <= size)
      {
         rxCount = bytesInBuf;
         USBCDC_receiveData(currentPos,rxCount,intfNum);
         currentPos += rxCount;
      }
      else
      {
         rxCount = size - (currentPos - dataBuf);
         USBCDC_receiveData(currentPos,rxCount,intfNum);
         currentPos += rxCount;
         return (currentPos - dataBuf);
      }
    }
    return (currentPos - dataBuf);
}
```

Figure 32. 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.

The same commentary applies to `hidReceiveDataInBuffer()`.

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

    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 33. 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. The endpoint buffers only store 128 bytes, and if the host tries to send more data before the existing bytes have been fetched, the device will begin NAKing the host (refusing to accept more data).

A good application of the `xxxReceiveDataInBuffer()` approach might be 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.

11.11.2 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. The API automatically moves all received data to the user buffer as it's received. When the application wants to, it can go to the user buffer and process 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.

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_setup();

    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;
           .
           .
           default:;
        }
     }
}
```

Figure 34. 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.

11.11.3 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.

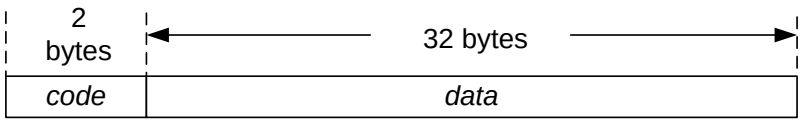


Figure 35. Command Packet

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

    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 36. 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 assigns 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.

11.12 Tips on Invoking BSL While USB is Active

Detailed information on BSL is documented in two TI-MSP430 documents:

- *MSP430 Programming via the Bootstrap Loader (slau319)*
- *Creating a Custom Flash-Based BSL (slaa450)*

This section discusses the steps to be taken to invoke BSL while USB is operating.

If the USB stack is active and the target device is in CDC, MSC or HID mode, the USB stack should be disconnected first before BSL is invoked via software. BSL is invoked when the program counter jumps to memory location 0x1000.

The following example shows the sequence of events using the USB APIs:

```
void invokeBSL(){

   __disable_interrupt();

   USB_disconnect ();         //PUR high, disable VBUS interrupt

   USB_disable();             //Disable USB module, disable PLL

   __delay_cycles(500000);    //Delay (if required)

   ((void (*)())0x1000)();    //Jump to 0x1000 to start of BSL.

}
```

Figure 37. Invoking BSL

12 Debugging Tips

12.1 The Device Enumeration Process

If a USB connection fails to operate as expected, a common first step is to determine whether the USB 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.

12.1.1 Summary of the Enumeration Process

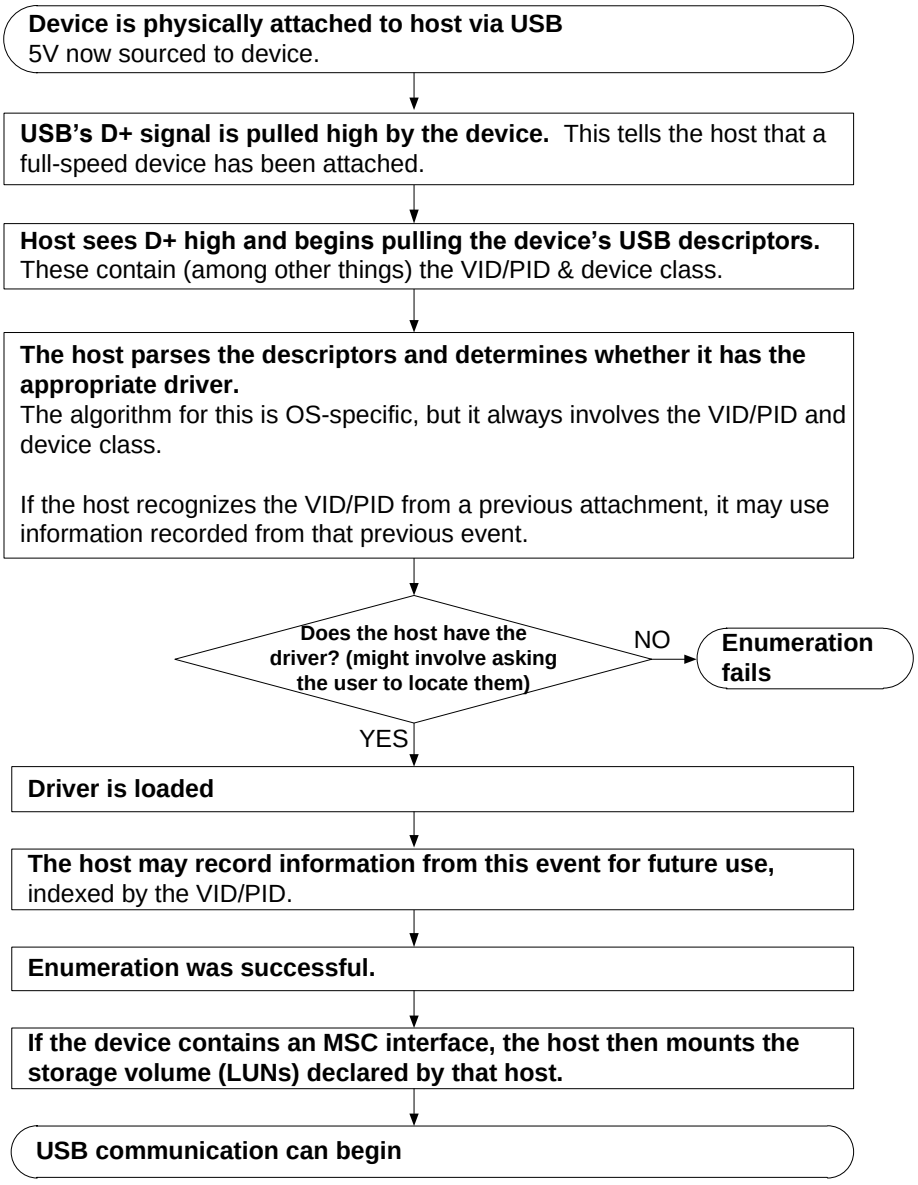


Figure 38. Basic Enumeration Flow

Pulling the D+ signal high via a pullup resistor is how a full-speed USB device tells the host it's available for enumeration. On an MSP430, this pullup is normally driven by the PUR pin. `USB_connect()` (perhaps via `USB_setup()`) activates the pullup; `USB_disconnect()` de-activates it.

The first time a device is attached, enumeration will probably take longer than subsequent attachments. This is because 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). On Windows, a message often is shown in the system tray, indicating that the device is "now ready to use".

Subsequent attachments may happen more quickly.

12.1.2 Determining Whether the Device Enumerated

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

On Windows, the first attachment of a device may result in activity in the system tray. A CDC interface, as discussed elsewhere, will result in an uninstalled device, until the user directs Windows to the INF file. Both of these indicate that the host at least saw the presence of the USB device. In turn, this means that the D+ signal was successfully 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.

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

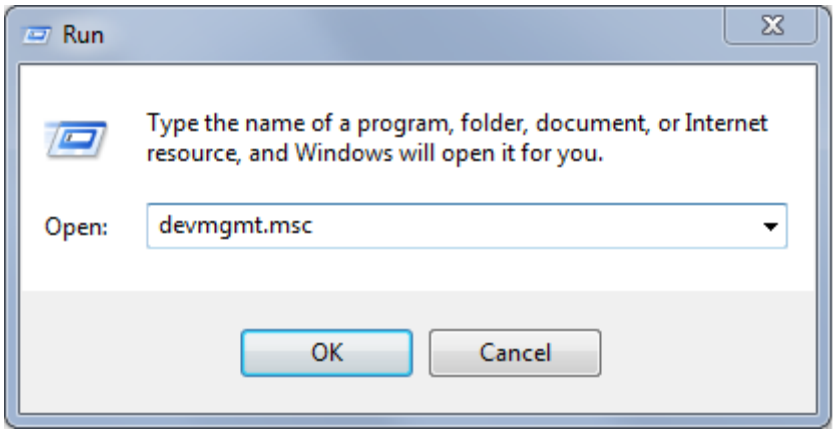


Figure 39. Starting the Windows Device Manager

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

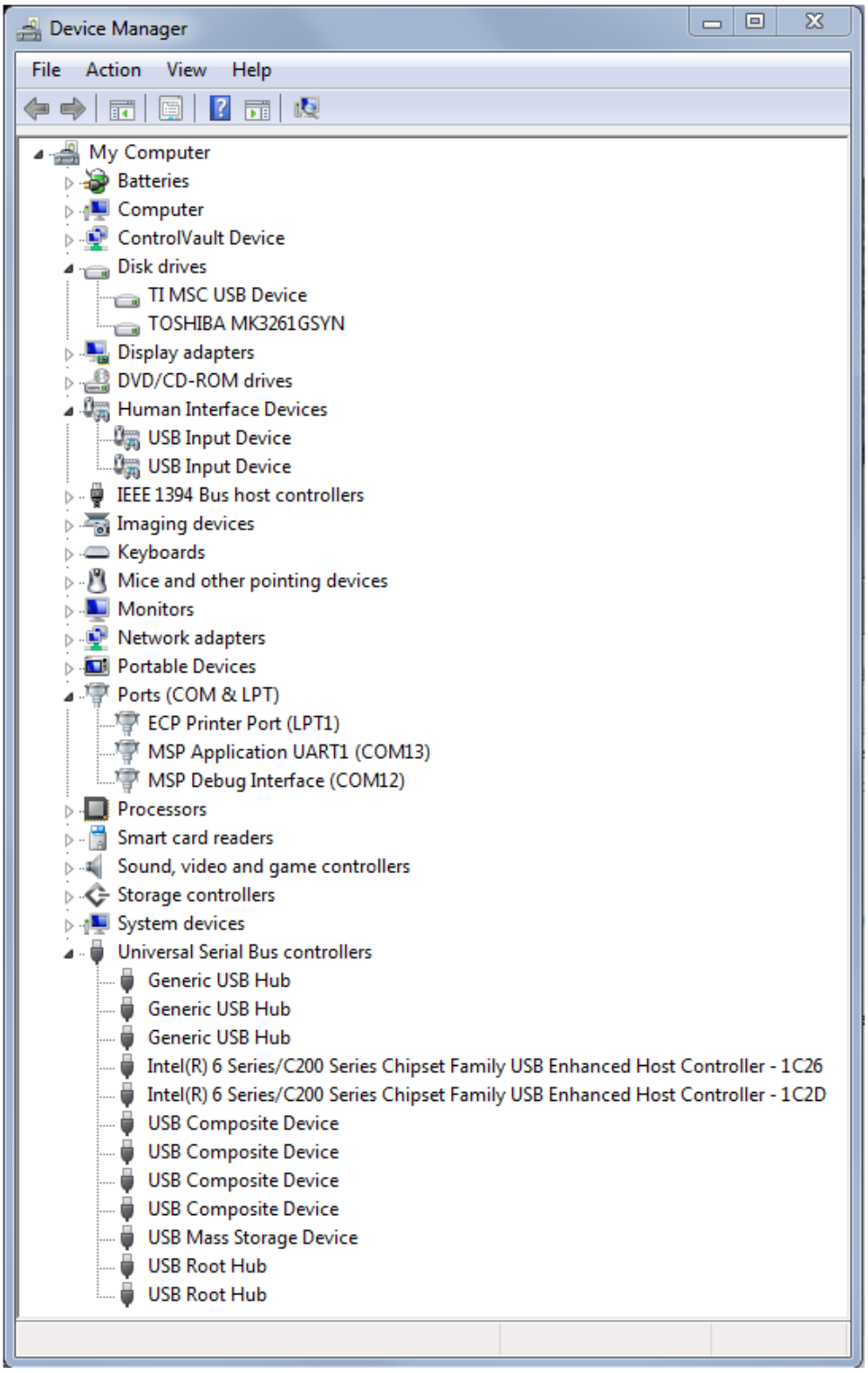


Figure 40. The Windows Device Manager

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

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

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. For example, the Mac OS provdies this in the "System Information" window.

12.1.3 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 via the PUR pin). This can be confirmed with a voltmeter or oscilloscope on the PUR or D+ pins. PUR is supposed to be pulled high when `USB_connect()` is called. It this isn't happen, it could be a problem with hardware (a short to ground), or perhaps software didn't call this function as expected.

12.1.4 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.

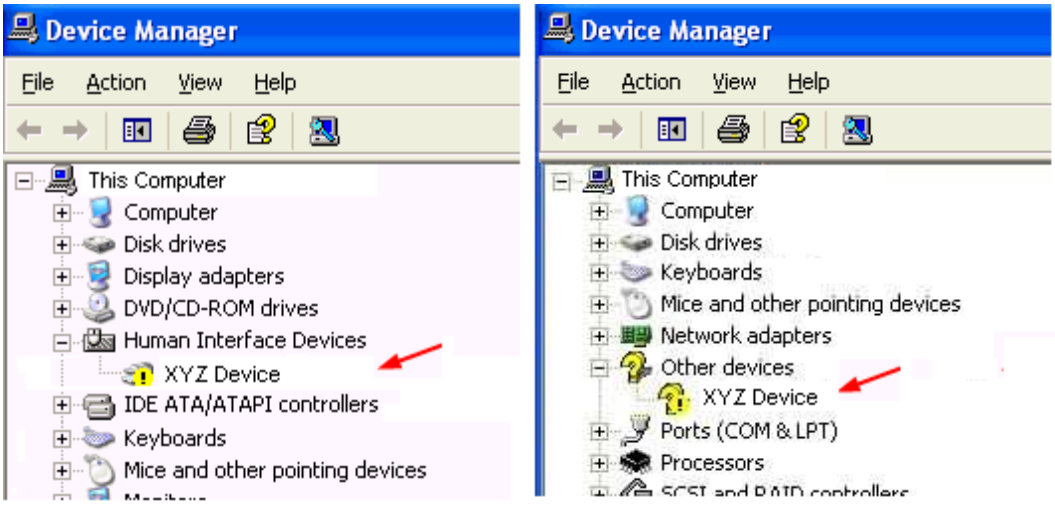


Figure 41. Failed Enumeration Results in the Device Manager

12.2 Common Problems

12.2.1 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". 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. It's best not to probe on the XT2 oscillator directly; instead, send XT2 to SMCLK or ACLK, activate the output pins for these clocks, and probe these output pins. (See Sec. 11.2 for more information on configuring the clocks.)

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. 11.3).

Finally, the API can only do its job if it's 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. 11.6.)

12.2.2 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.

If none of the 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. 12.3.

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. 12.3..

12.3 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. There’s no reason for USB descriptors for a given VID/PID to change, once a product is released.

During development, however, the USB descriptors may change as the developer modifies the device toward the final goals. It’s therefore easy for the developer’s host machine to see multiple descriptor sets for the same VID/PID. As a result, unless the developer remains conscious of the possibility of conflicts, problems are likely to occur on this host machine. Handling this is part of USB development.

One way to deal with this is to operate by this rule: if the descriptors change, increment the PID to a new value. 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 so this must match the one reported by the device in order to be recognized. The Descriptor Tool automatically generates the INF file alongside the USB descriptors; so in short, the files generated at a given time should be used together.)

Another approach, which doesn’t result in registry clutter, is to uninstall/install the device anytime the descriptors are changed for a given VID/PID. This will usually force the OS to discard old information and re-acquire it.

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

13 Using the API with RTOSes

Until now, the API has been shown as being called from a main loop, rather than with real time operating systems (RTOS). A main loop approach is accessible to the entire range of MSP430 USB customers. It effectively demonstrates the API. It also avoids the fact that there are many different RTOSes the developer may wish to use, and there's no way to demonstrate all of them. Further, many USB API applications don't require an RTOS.

However, the USB API fully supports usage with RTOSes, including:

- TI's SYS/BIOS
- Micrium's uC/OS-II and III
- FreeRTOS

This section contains guidance for using the API with RTOSes. The application examples also include two that demonstrate use with SYS/BIOS (examples #CHM2 and #CHM3).

13.1 General Design Considerations

When using the API with an RTOS, it's often necessary to call RTOS API functions out of the USB event handlers, for the reason of posting semaphores, sending messages to tasks etc. However, the event handlers are called within an ISR context, and this may be a problem for many RTOSes. Specific changes to the ISR are often required. In the case of the USB API, this means the developer may need to directly modify the USB ISR in UsbIsr.c.

Because the API is not fully re-entrant, there is a restriction with respect to the relationship between USB interfaces and tasks: a given USB interface should not be accessed by more than one task; doing so could result in conflicts. The exception to this is a multi-LUN MSC interface, where each LUN is allowed its own task, if desired.

If the USB device contains an MSC interface, then `USBMSC_poll()` needs to be called as part of the idle loop or task.

13.2 Using the USB API with TI's SYS/BIOS

SYS/BIOS is an RTOS freely-available from TI. It supports a wide range of embedded processors, including the MSP430. It provides a wide range of services, including:

- Pre-emptive multitasking
- Hardware abstraction
- Real-time analysis
- Memory management
- Configuration tools

SYS/BIOS is very scalable and can be configured to fit into memory-constrained MCUs. It's integrated into Code Composer Studio (CCS), making it very simple to create SYSBIOS projects within CCS. All SYS/BIOS tools are integrated into CCS as Eclipse plugins. (For MSP430-specific features and examples please visit the SYS/BIOS for MSP430 wiki page.)

This section discusses considerations for using the USB API with SYS/BIOS.

13.2.1 SYS/BIOS Thread Types: Hardware Interrupts (Hwi)

Hardware interrupt (Hwi) threads are essentially interrupt service routines (ISRs), and have the highest priority in the RTOS. They can be triggered by device peripherals, or externally. When an interrupt gets triggered, the CPU vectors into the ISR. As with all MSP430 interrupt handling, interrupts are automatically disabled within the ISR; the ISR will run to completion.

SYS/BIOS contains a Hwi module that can be used to manage ISRs. Handling of the interrupt vector is re-directed into the Hwi module, which then calls an application-defined function to handle the interrupt. This abstraction makes it possible for the interrupt handler – now an ordinary function – to call SYS/BIOS API functions. (If calling SYS/BIOS API functions isn't required, then the application may handle the ISR vector directly.)

With this in mind, in an application incorporating the USB API with SYS/BIOS, it's often valuable to make SYS/BIOS API calls within the USB event handlers. Since these handlers are called within the USB ISR, the aforementioned abstraction is needed. The USB ISR function `iUsbInterruptHandler()` needs to be made into an ordinary function, rather than a vector; and this function needs to be registered with the Hwi module.

The Descriptor Tool aids this process. The Tool always generates the USB ISR function, in the file UsbIsr.c. A checkbox in the Tool's MSP430 view results in the USB ISR being declared as an ordinary function.

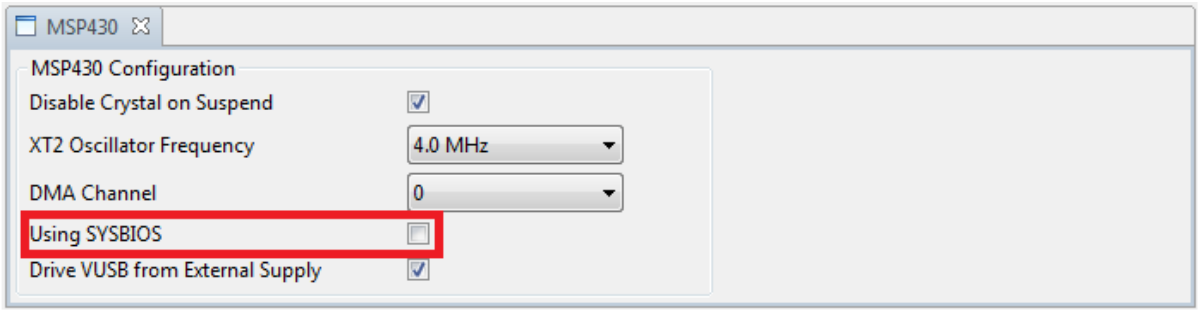


Figure 42. SYS/BIOS Checkbox in the Descriptor Tool

The application then needs to register `iUsbInterruptHandler()` with the SYS/BIOS Hwi module, using the following lines in the SYS/BIOS configuration script.

```
/* Create a Hwi Object and plug in the USB interrupt handler */
var hwi0Params = new Hwi.Params();

/* Have to make sure USB interrupt wakes up CPU */
hwi0Params.keepAwakeEnabled = true;

Program.global.hwi0 = Hwi.create(51, "&iUsbInterruptHandler", hwi0Params);
```

This enables the use of SYSBIOS API calls such as `Semaphore_post()`, `Event_post()`, `Swi_post()` within the USB event handlers.

The USB API package provides two examples that use SYSBIOS: #CHM2 and #CHM3. See the Examples Guide in the USB Developers Package for details.

Two other important configuration parameters for the Hwi module are given below. The first -- `Hwi.dispatcherAutoNestingSupport` -- needs to be set to false, in order to prevent interrupt nesting. (This is actually the default setting for MSP430, but worth double-checking, as an incorrect setting could result in a stack overflow.)

The other configuration is that `Hwi.fillVectors` needs to be set to false. This is because most applications have interrupts managed by the application, and the Hwi module should not generate duplicate vectors.

```
/* Do not allow interrupts to nest */
Hwi.dispatcherAutoNestingSupport = false;

/* Do not automatically fill un-used vectors. */
Hwi.fillVectors = false;
```

13.2.2 SYS/BIOS Thread Types: Software Interrupts (Swi)

The Swi module in SYS/BIOS provides *software interrupt (Swi)* capability. Software interrupts are patterned after hardware ones, but are scheduled by application software (using a SYS/BIOS API call such as `Swi_post()`), rather than generated from a hardware-based trigger. SYS/BIOS Swi threads have lower priority that Hwi threads, but higher priority than tasks.

Swi threads can be assigned a relative priority; lower priority Swi threads are pre-empted by higher priority Swi threads. The number of Swi priorities supported by SYS/BIOS is derived from a configuration parameter for the Swi module. Swi's are always pre-empted by Hwi threads.

Swi threads are suitable for handling application tasks that occur at slower rates, or are subject to less severe real-time deadlines than those of Hwis.

When using SYS/BIOS with the USB API, use of Swi's is optional. As discussed in Sec. 13.1, a USB interface shouldn't be handled by more than a single Swi thread.

The #CHM3 example demonstrates use of SYS/BIOS Swi threads.

13.2.3 Tasks (Task)

The Task module in SYS/BIOS provides tasking capability. Tasks have lower priority than either hardware or software interrupts.

The Task module dynamically schedules and pre-empts tasks, based on the task's priority level and the task's current execution state. This ensures that the CPU always executes the highest-priority available task. There's a maximum of 32 priority levels available for tasks, configurable by a parameter in the Task module.

The lowest priority level – "0" -- is reserved for running the idle loop. It cannot be used by the application.

To enable switching between tasks, SYS/BIOS maintains a run-time stack for each Task thread. On the MSP430, this task stack uses approximately 100 bytes.

When using SYS/BIOS with the USB API, the use of Tasks is optional. As discussed in Sec. 13.1, a USB interface shouldn't be handled by more than a single task thread.

The USB API provides the #CHM2 example that using SYS/BIOS Tasks.

13.2.4 Idle Thread

The idle thread runs continuously in SYS/BIOS whenever no Hwi, Swi, or Task is running. Idle functions are used to poll non-real-time devices that do not (or cannot) generate interrupts.

Idle functions all run at the same priority, sequentially, and in the same order in which they were created. An idle function runs to completion, before the next idle function starts running. When the last idle function has completed, the Idle Loop returns to executing the first idle function .

The idle thread can be used to contain all the USB functionality, in place of the main() function shown in this Programmer's Guide. This is demonstrated in the provided examples; the idle thread contains the forever while loop which checks the USB connection state, and puts the CPU into a low power mode whenever needed.

For applications using an MSC interface, the `USBMSC_poll()` should be called from the idle thread.

14 For More Information

Additional resources are shown below.

Note that the MSP430 method of organizating device information is to place architectural information – which is common to an entire family – is located with the family user's guide. Any device-specific information, and parametric information, is placed within the datasheet. This reduces the overall length of documentation, in an efficient manner.

General	
USB Developers Package Download	API software, examples, tools, and documentation for implementing USB on the MSP430. (Contains this Programmer's Guide.)
MSP430 landing page	General MSP430 information
MSP430 USB landing page	Information specific to doing USB on the MSP430
MSP430F5xx/6xx Family User's Guide	Architectural information about all devices in the F5xx family (including all the USB-equipped MSP430 derivatives).
Product Pages for USB-Equipped MSP430 Derivatives	
MSP430F552x MSP430F550x/5510 MSP430F563x MSP430F663x MSP430F565x MSP430F665x	Device-specific information for USB-equipped MSP430 derivatives (including datasheets).
USB Specifications	
USB 2.0 Specification	Specification for USB 2.0
Communications Device Class (CDC) Specifications	Specifications related to the Communications Device Class
Human Interface Device (HID) Class Specifications	Specifications related to the Human Interface Device class
Mass Storage Class (MSC) Specifications	Specifications related to the Mass Storage Class

Appendix A. Glossary

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" or "peripheral". This is a logical or physical entity on the bus containing one or more *USB interfaces*. It possesses one upstream-capable USB connector. The system being built with the MSP430 is the USB device.
- **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, and USB transfers happen through the pipe, between the host's endpoint and the device's 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.
- **Control Transfers:** One of four data transfer types on the USB bus. Control transfers are used for functions of USB management. Such transfers are called USB device requests: commands from the host, and responses from the device. This takes place on the control endpoint, which is always endpoint zero. Control transfers are used during USB enumeration. Individual USB interfaces may also use them for control functions, but generally send/receive data over the other endpoints, which are designated for bulk/interrupt/isochronous transfers.
- **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 suspend:** A low-power state imposed on USB devices by the host. When a USB device has been suspended, the amount of power it's allowed to draw from the host via the VBUS line within the USB cable is restricted. A USB device detects the signal to suspend by

sensing 3ms of inactivity on the bus. (Typically the host sends a Start of Frame packet every 1ms, thus a suspend occurs when SOF packets are no longer sent.)

- **USB resume:** After suspending a USB device, the host can later resume it to full operation. A resume is, effectively, the resumption of Start of Frame packets. Once a resume has been triggered, the device has 10ms to become fully operational.
- **VBUS:** The power rail within a USB cable. Through VBUS, the host can supply power to the USB device.
- **PUR:** After a USB device has become physically connected to the host, it can logically connect by pulling the D+ signal high (on full-speed devices) through a pullup resistor. The MSP430 pulls this resistor high via the PUR pin, controlled by software.
- **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.
- **HID report.** In a HID USB interface, the host reads/writes *reports.* A report is smaller than a USB packet (<64 bytes, in full-speed USB), and is organized according to a format defined by a *report descriptor* during enumeration.

API Stack Definitions

- **API Stack:** The USB API library provided by TI for the MSP430.
- **API Space:** The actual API source files. The API space is generally meant to be left unedited, although advanced users are free to change it, as needed.
- **Application Space:** An application is needed, to make calls to the API. This application exists in the *application space*. It is owned by the software developer. 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.
- **HID-Datapipe Interface:** HID interfaces in this API can either be *datapipe* or *traditional.* Datapipe eases the creation of general-purpose applications over HID. Defining features include use of a special, very simple report descriptor, and accessing the interface via the HID-Datapipe function calls.
- **HID-Traditional Interface:** This is any HID interface that isn't HID-Datapipe (see above). Mice, keyboards, and any HID interface with a custom report format are traditional HID interfaces.
- **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. The term generally refers to any

interface other than HID-Traditional, where the primary unit of information interchange is the *HID report.*

- **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.

General Definitions

- **COM port:** A software mechanism historically was associated with RS232 serial port interfaces. As RS232 was replaced by USB, Bluetooth, and other interfaces, the software mechanism was continued in form of a *virtual* COM port, implementing the same unformatted UART-style communication over these other interfaces. The term *COM port* is specific to the Windows group of operating systems; but the MacOS, Linux, and other operating systems provide similar mechanisms. The term “COM port” is used throughout this document to refer to all of them.