TEXAS INSTRUMENTS

July 2009

MSP430 USB Communications Device Class (CDC) API Programmer's Guide

MSP430

ABSTRACT

The MSP430 USB CDC API implements the USB's Communications Device Class on the MSP430. On most USB hosts, this class is associated with the "COM port" software mechanism. The API is designed for ease-of-use, particularly when used as part of the overall suite of USB tools provided by TI for the MSP430.

This Programmer's Guide augments the application examples that accompany the API.

Preliminary

Contents

1 Introduction 3
2 The MSP430 USB Tool Suite 3
3 How the API Relates to the System 3
4 USB States/Events and How They Relate to the API 12
4.1 Detection of the Host via VBUS 13
4.2 Enumeration 14
4.3 Failed Enumeration 14
4.4 Suspend/Resume 14
4.5 Remote Wakeup 15
4.6 Removal from the Bus 16
5 The API's Data Transmission/Reception Concepts 17
5.1 Introduction 17
5.2 Overview of Send/Receive Operations 17
5.3 The Lifecycle of an Operation 19
5.4 Background Processing of Send/Receive Operations 22
6 API Function Calls 25
6.1 MSP430 USB Module Management 25
6.2 USB Connection Management 28
6.3 CDC Management and Data Handling 32
7 Event-Handling 36
7.1 The Relationship between Interrupts and Events 36
7.2 Waking from Event Handlers 37
7.3 Using *USB_setEnabledEvents()* 37
7.4 Event Handler Functions 39
8 Configuration Constants 43
9 Practical Matters: How to Write USB Programs with the API 45
9.1 Considerations When Designing a USB Program 45
9.2 Main Loop Framework 47
9.3 Recommended Constructs for Send Operations 49
9.4 Recommended Constructs for Receive Operations 55
9.5 Send/Receive Construct Functions: Summary 60
10 References 63

Preliminary

1 Introduction

The USB CDC API (USB Communications Device Class Application Programming Interface) for the MSP430 is a turnkey API that makes it easy to implement a simple data connection over USB between an MSP430-based device and a USB host. It abstracts the user from the USB protocol and is designed to be intuitive while providing a wide degree of configuration capability.

To the USB host, the connection appears as a virtual COM port, which is a very simple and commonly-used interface. To the MSP430's software, it appears as a data interface that is accessible with simple API calls, such as "open/close the connection", "send/receive data", etc.

The user shouldn't need to modify the API source. However, the source is made open and available to assist in understanding the entire system, if needed.

Application examples are included with the API that demonstrate the concepts described in this document. This Guide can serve as a reference to those examples.

2 The MSP430 USB Tool Suite

Preliminary

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

- *MSP430 USB Descriptor Tool:* automatically and quickly generates reliable customized *descriptors.c/h* files for use with the API. It saves the developer's time and reduces the chance for errors.
- *MSP430 USB HID API:* an API similar to this one, for implementing the Human Interface Device class
- *MSP430 HID Windows API:* an API for use with the MSP430 HID API. It simplifies creation of a HID device for designers unfamiliar with the PC implementation of HIDs.
- *MSP430 USB Device Firmware Update (DFU) Starter Project:* a Windows Visual Studio project for an application that downloads firmware to the MSP430, just by double-clicking. It uses the MSP430's on-chip USB bootstrap loader. Simply insert the new firmware image file and compile; or, if desired, modify it, or migrate the functionality into a larger application. (Demo app available now; starter project available August 2009)
- *MSP430 USB MSC API:* an API for implementing the Mass Storage class (Nov 2009)

Please check with TI for updated schedules.

Note:

References are made throughput this document regarding support for composite USB devices. However, v1.0 of this API stack only supports single-interface CDC devices.

3 How the API Relates to the System

3.1 COM Port and CDC Device Class Overview

The purpose of this API is the establishment of a COM port on the USB host, through which an application on the host can communicate with an MSP430 over USB. On the MSP430 side, the PC COM port is associated with a simple data interface, through which data is transceived with the host. Calls are available to accomplish this, at a level of complexity similar to that of a basic UART.

(The term *COM port* is specific to the Windows group of operating systems, but the MacOS, Linux, and other PC-oriented operating systems provide similar mechanisms. The term *COM port* will be used throughout this document to refer to all of them.)

The COM port software mechanism is a popular means of communicating with a peripheral from a PC, due to its simplicity. It was originally designed for communication over the RS232 port, but it has now grown beyond these hardware limitations. Often today it is used in the form of a "virtual" COM port operating over USB or Bluetooth, interfaces which by now have mostly replaced RS232 on the PC back panel.

Historically, COM ports were used in telecom applications, including modems and early networking applications. Although these applications for COM ports have largely been replaced by other uses, this telecom influence exists to this day. For example, programs whose primary function is to interface with COM ports are still referred to as "terminal" applications, a term derived from these early roots.

Preliminary

One means of establishing a virtual COM port interface over USB is with the Communications Device Class (CDC) protocol, which was established by the USB Implementers' Forum. The major OSes have already included support for this protocol as part of their basic installations; and so for most applications, the end user need not install custom kernel drivers. This leads to greater system stability, less hassle for the end user, and considerably less hassle for the OEM, in terms of development cost and end user support.

The scope of the CDC spec is much wider than virtual COM ports, supporting a wide variety of communications equipment. As a result, this API supports only a subset of the CDC spec. Specifically, it supports the Abstract Communication Model (ACM) of the PSTN subclass. The ACM provides for a control mechanism using common V.250 AT commands. This is sufficient to establish a fully-functional virtual COM port interface.

By default, a single data link is established, in which one USB interface of class CDC on the MSP430 is interfaced to one COM port on the PC. If it is desired to establish more than one of these interfaces (another CDC interface linked to another COM port), it can be accomplished by using multiple CDC interfaces within a single piece of USB hardware. A USB device with more than one interface is referred to as a *composite* USB device. This API supports up to three CDC interfaces.

3.2 Stack Organization

A code stack using this API is shown in Fig. 1.

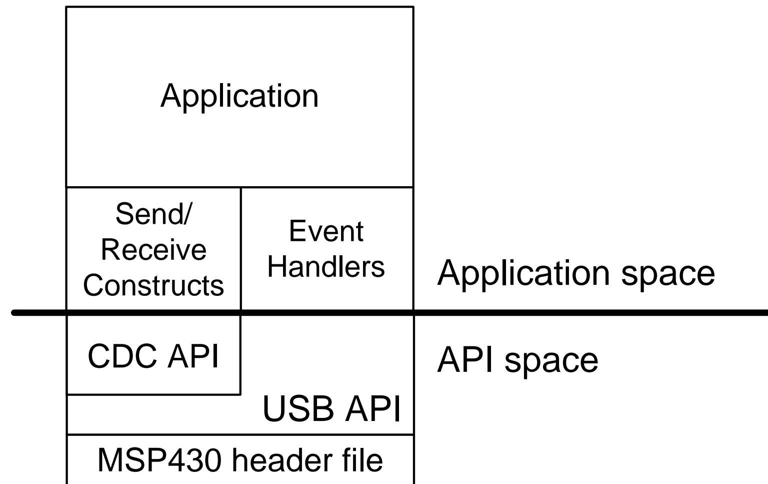


Figure 1. CDC API Software Stack

The USB functionality is fully provided by the API, allowing the user to focus on data handling rather than USB protocol. Resource usage is clearly identified so that the developer knows what resources are available for the application.

Some API calls are common to any USB interface, such as initialization, connecting to the USB, etc. These are located within the USB API layer. Other calls are specific to the CDC; the CDC layer interfaces with the USB API to actualize the CDC protocol. The USB API is shared among all MSP430 USB device class API stacks distributed by TI – for example, the HID API – which eases the development of composite devices with multiple device class APIs.

Interrupt handling (that is, the USB interrupt service routine) is contained within the API, as opposed to user-defined application space. However, the API provides to the application a means of responding to interrupts through *events.* Events can be thought of by the developer as the same as an interrupt, and in most cases are called out of the USB ISR. Handler functions for each event are defined by the API, and these are considered to be in application space for the developer to customize for the application. For example, when the host suspends the device, or when a send operation is complete, the event gives the application a chance to respond. See Sec. 8 for more information about event handler functions.

The API also provides pre-defined send/receive constructs. In most cases, the application is best served by performing its sending/receiving using these constructs, rather than direct calls to the API. These constructs represent the most robust ways to use the API's send/receive calls.

Preliminary

3.3 Source File Organization

Source and header files that comprise the API stack are shown in Table 1.

Table 1. Files in the API Stack

	User-Modify?	Filename	Description
Application-specific	Yes	*main.c*	User application.
		usbcdc_constructs.c	Contains recommended constructs for send/receive operations. The functions here reflect the approaches described in Sec. 9.3 and 9.4.
		usb_eventHandling.c	Event-handling placeholder functions.
		descriptors.c/h	*Descriptors.c* contains data structures that define the USB descriptors the device will report to the host during enumeration. A default descriptor set is provided, which can be customized directly or with the *MSP430 USB Descriptor Tool*. *Descriptors.h* contains constants that support the descriptors, and also contain the configuration constants, as described in Sec. 8
API stack	No	*usbcdc.c*	CDC-related functions.
		usb.c	All non-CDC and non-HID, USB-level functions; descriptor handling; API-defined calls made from ISR vector.
		usbisr.c	USB interrupt service routine handler and related functions.
MSP430 header file	No	I.e., *MSP430xx55x2.h*	Standard header file for the MSP430 device derivative being used. This is included in the software development environment, outside this API.

Preliminary

3.4 Use of MCU Resources

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

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.

Two DMA channels must be assigned to the API, one for transmit and one for receive. The channels are assigned using the configuration constants USB_DMA_TX and USB_DMA_RX. (See Sec. 8.) All interfaces share the same pair of DMA channels.

The API uses approximately 4.5K bytes of code flash and 78 bytes of RAM.

3.5 Descriptors

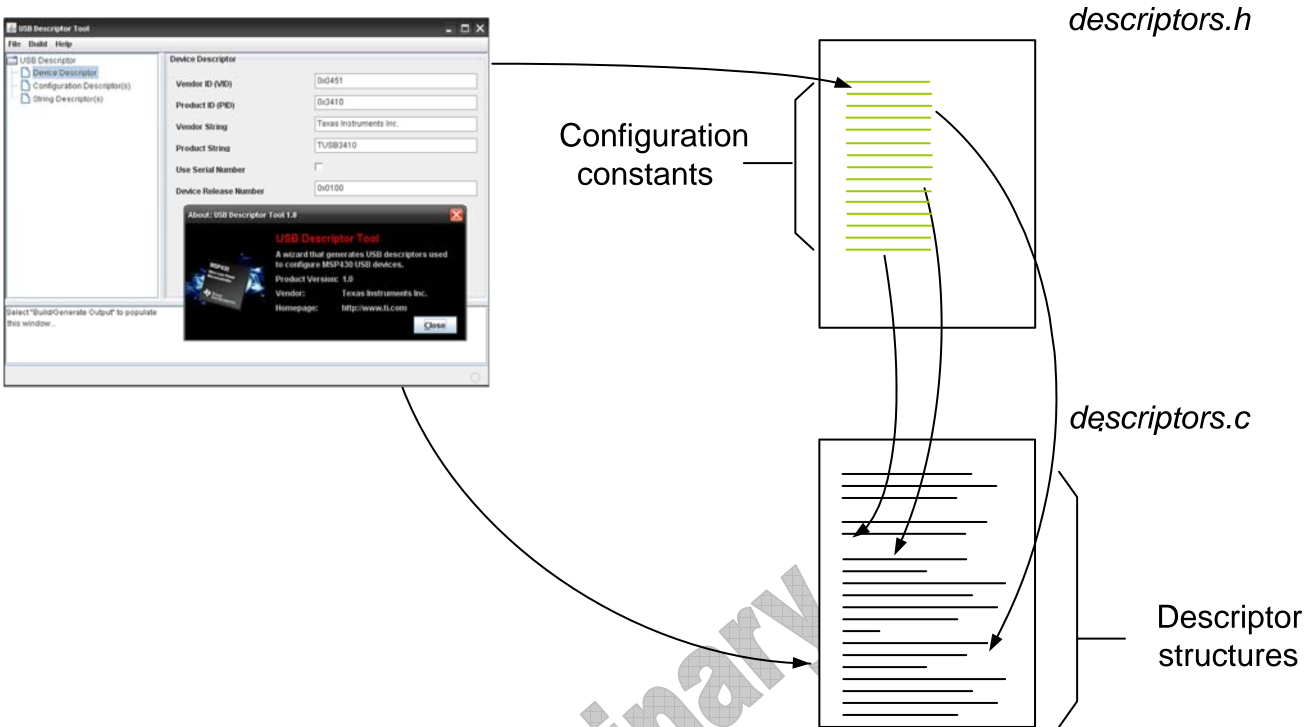
Every USB device contains *descriptors*, which communicate to the host the device's class, as well as what its capabilities are. The USB descriptors reported by the CDC API reflect the requirements of the CDC specification for the feature subset implemented by the API. The descriptors are defined in the file *descriptors.c*.

By default, the API's descriptors report the device with TI's vendor ID, a product ID specific to this API, and TI-specific vendor/product strings. (See the section below for information about identifying VIDs/PIDs.) This is acceptable for development, but for production purposes they should eventually be customized with information specific to the vendor and product. The descriptors have been designed with a certain amount of abstraction, such that the application developer should not need to modify them directly. This makes development faster, and avoids the chance of introducing errors.

This abstraction is accomplished at two levels. The first is that most of the application-specific values have been abstracted into configuration constants. This allows modification of the constants rather than the descriptors themselves. The configuration constants also control the high-level functionality of the API stack. See Sec. 8 for a complete list of configuration constants.

The second level is that TI provides a tool called the *MSP430 USB Descriptor Tool.* This tool automatically creates *descriptors.c/h* files that replace the default ones included with the API. It does its work primarily by modifying the configuration constants based on GUI input, and also by building the string descriptor according to the strings input by the user. In the case of a composite device, it also manages the entire descriptor tree based on the user's inputs.

Preliminary



Preliminary

Figure 2. Descriptor Modification Using Configuration Constants and the Descriptor Tool

Generally speaking, it should not be necessary to modify the configuration constants directly, rather they can be controlled via the Descriptor Tool.

NOTE: v1.0 of this API stack does not support composite devices

3.5.1 Choosing a Vendor ID / Product ID

A USB product is identified by a unique combination of Vendor ID (VID) and Product ID (PID). VIDs are unique to a particular vendor/OEM; PIDs are unique to a particular product/model sold by that vendor. If vendor X sells a product Y, then every physical instance of that product has the same VID/PID pair. VIDs and PIDs are both 16-bit values.

VID/PID pairs are important in how host operating systems identify the device during the process of enumeration. Once a USB device has been enumerated on a given host for the first time, the VID/PID pair is often the way in which the host recognizes that device thereafter, and it responds by loading the driver it associated with that device the first time it was attached. This is usually a valid assumption, even if the second attach is in fact a different physical device, since all physical instances of that product are more or less identical and should use the same driver. So even if another instance of the device is attached, the same driver should apply.

VIDs are assigned by the USB-IF to a particular company, for a fee. A vendor may contact the USB-IF to obtain one (www.usb.org). Alternatively, TI has a VID-sharing program, in which TI will license a unique PID from within TI's VID space to MSP430 customers, at no cost. Customers can contact TI for more information (www.ti.com/msp430).

PIDs are completely at the discretion of a vendor if it owns its own VID. Care should be taken not to duplicate PIDs within the company; a particular product should have a unique PID, by convention, and also by practicality: products of different device class types but sharing the same VID/PID could create driver conflicts if both devices are attached to a particular host machine.

3.5.2 Serial Numbers

VID/PID pairs identify a particular product or model within all USB products sold on the market. A *serial number* in conjunction with a VID/PID pair can identify a specific piece of USB hardware on the market. Unlike VIDs/PIDs, which are stored as fields in the device descriptor, a serial number is implemented as a string, in a string descriptor.

Since all instances of a product contain the same firmware image, storing a unique serial number creates some additional difficulty. It either must be programmed into the device at production time, or the silicon must have a unique ID built into it. Fortunately, this is the case with MSP430.

Preliminary

The API derives a unique serial number from the device ID that is unique to each physical MSP430 device. The reporting feature is fully automatic and easily controlled using the value of the configuration constant USB_STR_INDEX_SERNUM:

- If set to zero, no serial number will be reported.
- If set to a non-zero value, the API will generate and report a serial number. The non-zero value must be equal to the index of the serial number dummy string descriptor that is provided with the API. (The default value of this index is provided in the comments next to USB_STR_INDEX_SERNUM.)

If using the Descriptor Tool, this feature is easily controlled by using a checkbox.

If a USB device does contain a serial number, the operating system will probably use it to help identify that device and to differentiate it from other devices of the same VID/PID. This can allow system settings associated with a particular piece of hardware to “follow” that device no matter what port it's attached to. Without a serial number, the host cannot tell one instance of a given VID/PID from another. Since most hosts have multiple ports, it attempts to assign a ‘unique’ identifier based on what port it's attached to. As a result, moving that device to a different port makes it appear to the host as a different device, and any configuration information that was associated with that device will no longer be associated with it. Reporting a serial number solves this problem.

There is no way for software to control the device ID; it is hardwired in the silicon.

3.5.3 Configurations

The API's default descriptor set provides for a single USB configuration. By default, a string exists for the lone configuration descriptor, which should be customized using the Descriptor Tool.

3.5.4 Interfaces/Endpoints

By default, the API provides a single CDC interface. This results in a single COM port on the host. The endpoint usage reflects the requirements of the CDC specification, as implemented by major operating systems. It is shown in Table 2.

Table 2. Endpoint Usage

Endpoint	Default Endpoint	Purpose
Control/status input	IN0	Management Element
Control/status output	OUT0	
Interrupt input	IN2	Notification Element
Bulk input	IN3	Data Interface input
Bulk output	OUT3	Data Interface output

Preliminary

3.6 Power Management

The MSP430's USB module has an integrated LDO regulator that reduces the 5V VBUS power supply from the USB to 3.3V. The MSP430 can then be powered from the USB.

The MSP430's USB LDOs automatically activate when 5V bus power is applied to VBUS. The application of 5V bus power generates a VBUS-on event (*handleVbusOnEvent()*) from the API, if enabled.

The user is encouraged to use the MSP430's low power modes in the application, because maximizing the time spent in these modes can greatly increase power efficiency. They can be used as they are in any MSP430 application, with the one restriction that while enumerated on a USB host, the application must stay in LPM0 rather than entering LPM3/4.

In battery-powered systems, the engineer is encouraged to design the power system so that the USB device draws most of its power from the bus while VBUS power is available. This conserves battery power. It should be kept in mind that being attached to a USB host may keep the device active for longer periods of time than would occur for a mobile device.

3.7 Clock System

MSP430 devices supporting USB include a PLL that generates a USB clock from a high-speed crystal on either the XT1 or XT2 oscillator. A wide range of crystal frequencies is supported. The API function call must be told at what frequency the crystal oscillator is operating, so that it can configure the dividers in the PLL accordingly.

The API must be configured according to the chosen crystal frequency and the oscillator on which the high-speed crystal is configured to drive the PLL. This is controlled with the USB_PLL_XT and USB_XT_FREQ configuration constants, respectively.

3.8 Host Considerations

The CDC driver is supported natively by the three major operating systems – Windows, the Mac OS, and Linux. In this way, it is a well-supported and easy solution.

3.8.1 Microsoft Windows

Windows responds to the descriptors reported by this API by establishing a "COM port" link, similar to what it does with legacy RS232 hardware serial ports. This is typically referred to as a "virtual" COM port.

The most basic Windows application that can utilize a COM port is Hyperterminal, which is a part of any Windows installation. An MSP430 board loaded with this API, attached to a Windows-equipped USB host, will allow communication with Hyperterminal, once the proper COM port has been chosen by it.

The CDC class is supported by Windows 2000, XP, and Vista. Although the driver binaries are built into Windows, no INF file is provided within Windows for the driver. This means the OEM must provide an INF file with the hardware. In turn, this means that the end user needs to go through a device installation process in which he or she selects the new INF file.

Preliminary

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

A state diagram with regard to a device's interaction with the USB bus is shown in Fig. 3.

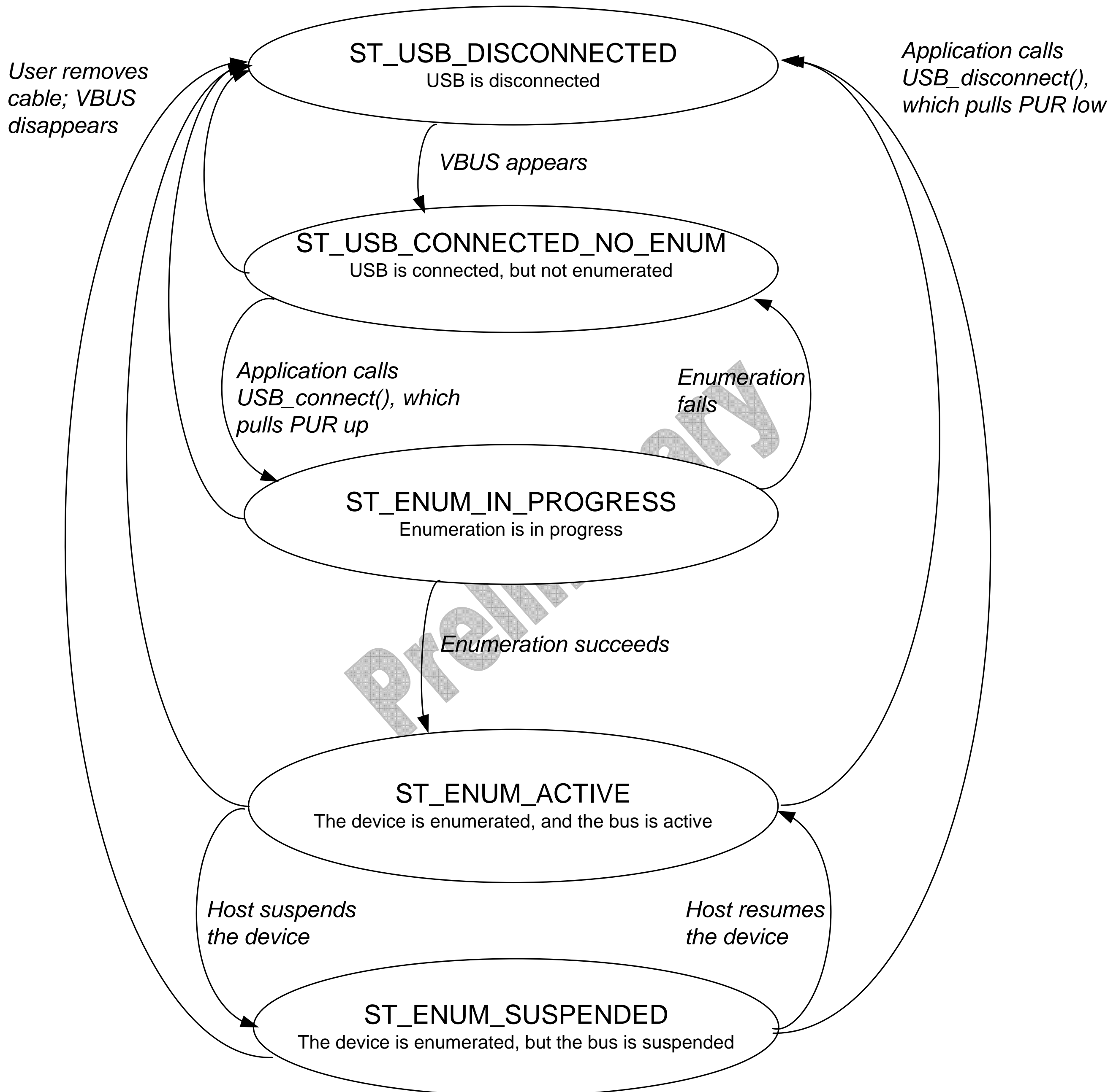


Figure 3. State Diagram Reflecting a Device's Interaction with USB

Preliminary

The states are defined in Table 2.

Table 3. USB State Definitions

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

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

- *VBUS*: 5V power from the host. If present, the application can assume a host is attached.
- *PUR*: PUllup Resistor. A USB device signals its presence to the host by pulling up the D+ signal through a resistor. The MSP430 implements this with the PUR pin.
- *Enumerated*: When a USB device has been successfully enumerated, it means the host has successfully interpreted the device's descriptors and has loaded the appropriate driver.
- *Suspended*: A host can suspend a USB device at any time, at which point no communication can take place, and power drawn from 5V VBUS is restricted to <500uA.

Preliminary

A function is provided, *USB_connectionState(),* which returns the state in which the connection lies (left column). Another function, *USB_connectionInfo(),* returns the lower-level information that defines the states (top row). Most applications are best served by using *USB_connectionState()* to direct program flow, but there might be occasional need to call *USB_connectionInfo()* as well.

A device is likely to spend most of its time in the disconnected, active, and suspended states, the proportions of which are dependent on the application. The other two are usually fairly transient.

The sections that follow elaborate on various points related to the states.

4.1 Detection of the Host via VBUS

A device can know an active host is present by sensing the availability of 5V on the VBUS signal. This information is used in the API's execution of *USB_connectionState()* and *USB_connectionInfo().*

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

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

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

4.2 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 pullup, and it is controlled by the PUR pin. As mentioned above, the default *handleVbusOnEvent()* handler activates this pullup by calling *USB_connect().*

When the host sees this pullup, 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.* (See Sec. 3.5.)

When enumeration is complete, a call to *USB_connectionState()* reflects this, and the system is ready to transfer data.

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

Preliminary

4.3 Failed Enumeration

If the device pulls PUR high and it does not lead to successful enumeration, *USB_connectionState()* will stay in the ST_ENUM_IN_PROGRESS indefinitely. This probably won't be a common occurrence, but it can happen in a variety of situations. For example, if the end user didn't fully complete a previous install process, it could corrupt the device's association with the driver within the host OS. Enumeration of the USB device may fail thereafter until the host situation is resolved.

The software designer should consider how the device should respond in this situation. Perhaps the device should be allowed to remain in this state until the user detaches the device from the host. A sophisticated approach would be to recognize after a period of time that the enumeration has failed, and make a call to *USB_disconnect().* This could be done with an incrementing counter, or with a timer. A difficulty with this approach is setting the timeout period, since in the case of initial enumeration, the enumeration period may be subject to how quickly the end user finishes the device installation. In other words, it could be several minutes or longer. Performing a disconnect too early risks corrupting the driver association on the host.

4.4 Suspend/Resume

At any point after successful enumeration, the host may choose to suspend the device. When a device has been suspended, it cannot communicate with the host until it has been resumed by the host (see the next section, regarding remote wakeup). It also has 10ms to move into a state where it consumes less than 500uA of power from VBUS.

The API handles suspend and resume somewhat automatically, as far as the USB module is concerned. Calls also exist for this purpose: *USB_suspend()* and *USB_resume().* These calls are made automatically from within the USB ISR, and so there is usually no reason for software to call these functions.

If the device is self-powered, it does not need to be concerned about the 500uA limit. If it does draw from VBUS, however, software must be prepared to put the USB device in a low-power state within the specified time. This might mean powering down other MSP430 peripherals, or other components on the board. A good place to handle this is the *handleSuspendEvent()* handler, which executes out of the USB ISR immediately after *USB_suspend()* is called. Alternatively, if USB_connectionState() is being polled, a return value of ST_ENUM_SUSPENDED can be used to trigger this function.

It is important that the software designer consider that a suspend can happen at any point in the code flow, and build the code so that it can exit from any event when the USB state has changed.

Resume is handled similarly. A call to *USB_connectionState()* will begin returning ST_ENUM_ACTIVE. The event handler *handleResumeEvent()*, if enabled, will also execute. Either can be used to trigger code that changes the state of the system in response to the resume event.

Preliminary

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.

4.5 Remote Wakeup

A remote wakeup event is a mechanism by which a suspended USB device can prompt the host to resume it. The host does not need to comply, but it may. A common example of remote wakeup is when a USB mouse is attached to a PC, and the PC goes into standby mode. In some configurations, the mouse may be able to wake the PC when moved. In this case, the mouse issued a remote wakeup event to the host, and the host responded by waking itself, and then resuming the mouse.

A device first must declare itself as capable of remote wakeup within its configuration descriptor. The API provides for this via the USB_SUPPORT_REM_WAKE configuration constant. When this is set to indicate support for remote wakeup, the host can choose to grant the ability for remote wakeup using a SetDeviceFeature(DEVICE_REMOTE_WAKEUP) request, or it can choose not to. Whether it does so is OS-dependent and also dependent on user configuration.

When an application wants to issue a remote wakeup, it can do so 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.

4.6 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). This generates a USB interrupt, which calls the *handleVbusOffEvent()* function. Code can be placed here to disable the USB module. Similarly, a call made to *USB_connectionState()* at this point will return ST_USB_DISCONNECTED.

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

Preliminary

5 The API's Data Transmission/Reception Concepts

5.1 Introduction

The API provides a simple scheme of exchanging data with the USB host, using commands such as "send data" and "receive data". It also allows transfer of data beyond the limits of a USB packet (that is, a maximum size of 64 bytes). In so doing, it abstracts the user from the USB and CDC protocols, handling these functions automatically.

For each interface descriptor in *descriptors.c*, there exists an *interface* within the API that can be accessed with API function calls. A single interface is associated with a single COM port on the PC. Multiple interfaces can be supported by the API, and all function calls related to sending/receiving data have a parameter *intfNum* that selects which interface is being referenced.

(Note: In v1.0 of the CDC API stack, only a single CDC interface is supported.)

Note: This section serves as a reference for how send and receive operations work, whereas Sec. 9.3 and 9.4 give practical information for how to use them. Those sections define recommended constructs, and functions that implement them. In most cases, applications are best served by making calls to these constructs rather than calling the API's functions (*USBCDC_sendData()* and *USBCDC_receiveData()*) directly. However, having an understanding of how send/receive operations work provides a deeper understanding of the API.

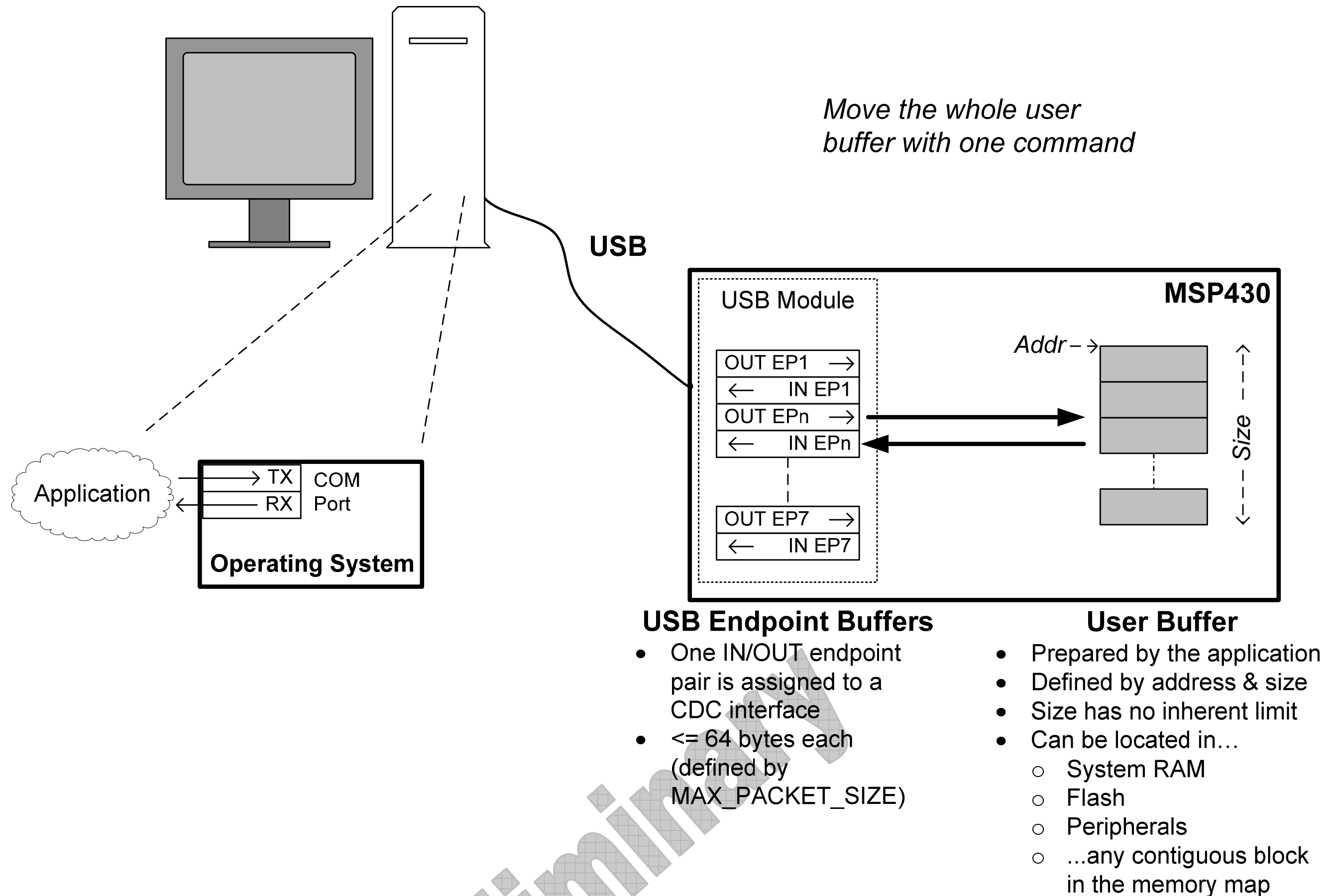
Preliminary

5.2 Overview of Send/Receive Operations

To send data to the host, the application first prepares a data source, called the *user buffer*. The term "buffer" implies a RAM source, but it can also be flash or peripherals – any contiguous block in the memory map. During a *send operation,* the API copies the data to the USB endpoint buffer in packetized chunks. After each packet is formed in the endpoint buffer, the packet is made available to the host. At the host's discretion and timing, it reads the data from the endpoint buffers. A send operation is initiated with a single call to *USBCDC_sendData().*

For receiving data, the action is similar. The application prepares a user buffer as a data sink. It can be any contiguous block in the memory map capable of handling the data. During a *receive operation,* as data is received by the host into the USB endpoint buffers for this CDC interface, the API copies it into the user buffer. A receive operation is initiated with a single call to *USBCDC_receiveData().*

These are shown in Fig. 4.



Preliminary

Figure 4. The Context of Send/Receive Operations

The API's *operation* concept frees the application software from having to be "on call" to move data every time a packet is transceived; rather the data is sent/received with a single call to *USBCDC_sendData()* or *USBCDC_receiveData().* After beginning the operation, these functions return execution to the calling function; the operation then runs in the background.

This is directly analogous to using the USCI module in conjunction with the DMA to implement serial communication via UART or SPI, where the DMA transfers a block of data to/from the USCI module as the transmit/receive registers become open. In that context, it's the DMA that manages the transfer in the background; in this context, it's the API.

The user buffer is defined by *address* and *size* parameters. A send operation completes when all *size* bytes have been sent to the host. A receive operation completes when all *size* bytes have been received from the host. Data is always moved sequentially from the start of the user buffer to the end.

A send or receive operation has no inherent size limit; this allows the application to focus on data transfer rather than packetization. It may be used with large sizes (that is, larger than the size of the USB endpoint buffers) or small sizes. There is no requirement that data align to packet boundaries. As long as a buffer of the specified *address* and *size* truly exists and can be read by the API, any size is valid.

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.

The size of the USB packets used to implement an operation is determined by the configuration constant MAX_PACKET_SIZE. In most cases, CDC implementations are best served by the maximum value of 64 bytes. If less data is sent, a smaller packet will be automatically used, thus not wasting bandwidth.

5.3 The Lifecycle of an Operation

Send and receive operations for a given interface do not interact with each other. That is, an interface may have one send operation and one receive operation – but not more than one of either – simultaneously.

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

5.3.1 Lifecycle of a Send Operation

The lifecycle of a successful send operation is shown in Fig. 5.

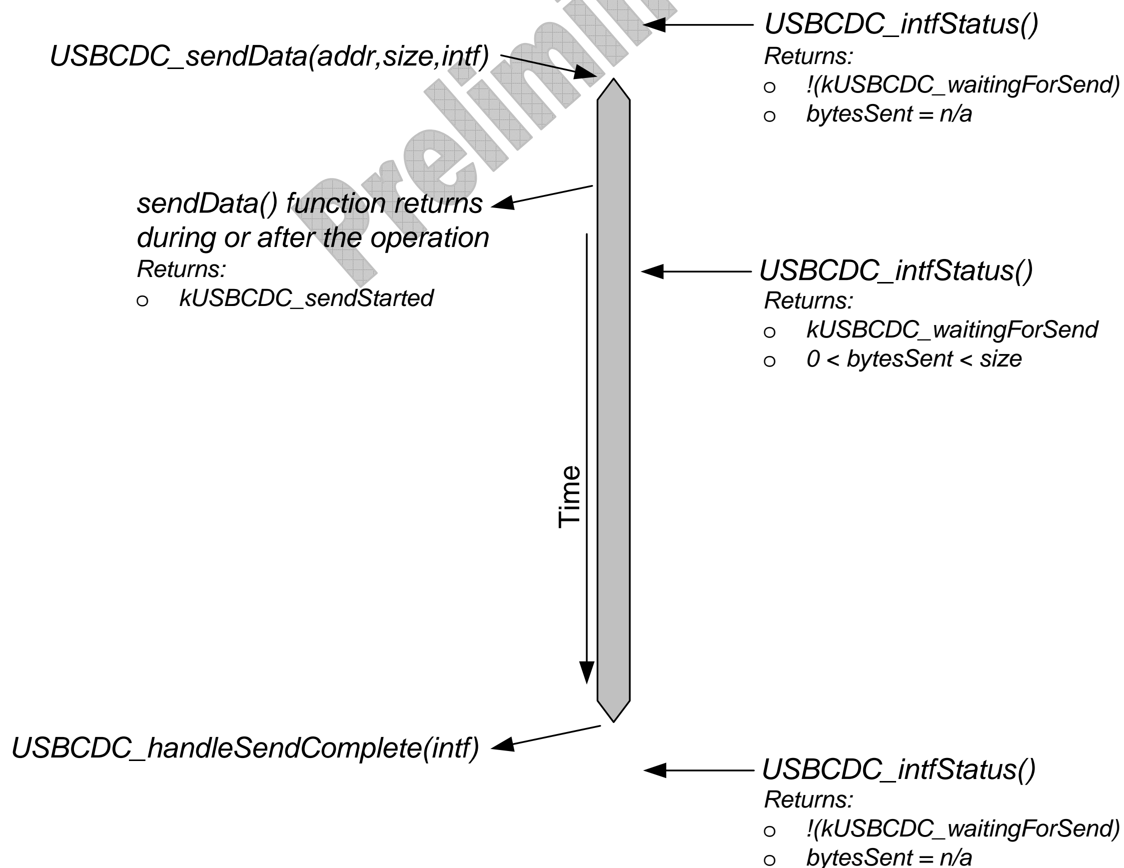


Figure 5. Successful Send Operation

Preliminary

As discussed earlier, a send operation is begun with a call to *USBCDC_sendData()*. The application can make its first call to this function any point it wishes after enumeration is complete. A successful call to this function returns *kUSBCDC_sendStarted.*

If a call to *USBCDC_sendData()* is made while a previous send operation is underway, it will immediately return with a value of *kUSBCDC_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 *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 *USBCDC_sendData()* returns *kUSBCDC_sendStarted,* software should be aware the operation might still be open. Therefore, any subsequent calls to *USBCDC_sendData()* should check to ensure that no previous operation is underway. (See the next section.)

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

Preliminary

5.3.2 Lifecycle of a Receive Operation

Similarly, a receive operation is begun with a call to *USBCDC_receiveData()*. The lifecycle of a successful receive operation is shown in Fig. 6.

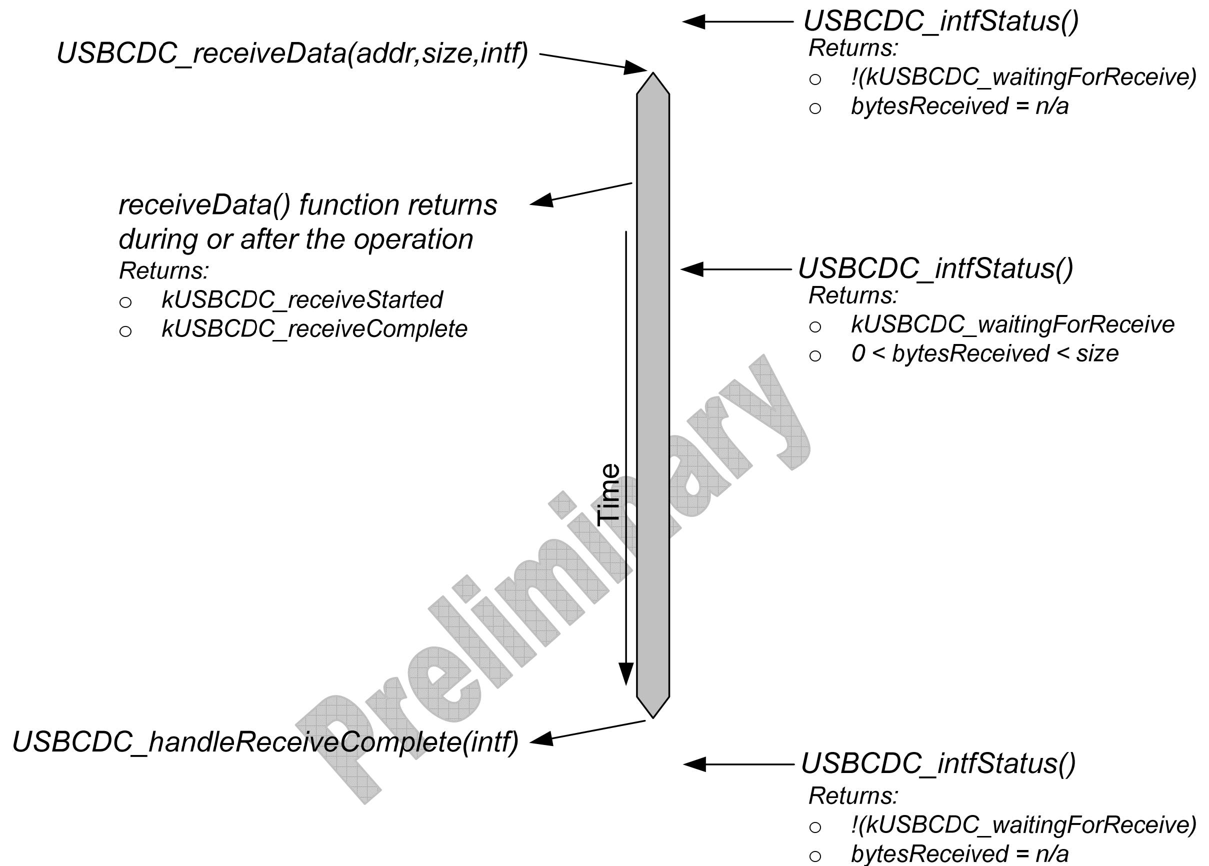


Figure 6. Successful Receive Operation

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

If a call to *USBCDC_ receiveData()* is made while a previous receive operation is underway, it will immediately return with a value of *kUSBCDC_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 *USBCDC_handleReceiveComplete().* User code may be placed there, including the setting of a flag to signal 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 clogged. Opening a receive operation gives the incoming data a place to go.

If this situation occurs, the API makes a call to *USBCDC_handleDataReceived().* In addition to opening an operation, software has the option of calling *USBCDC_rejectData().* This flushes the USB buffer rather than giving it an operation by which to move it. The "pipe" becomes unclogged again; but the data that was in it is lost.

5.4 Background Processing of Send/Receive Operations

This section discusses the API's background execution from a conceptual standpoint. Sec. 9.3 and 9.4 provides pre-built functions that can be used in an application to take advantage of it.

Operations often remain open after the call to *USBCDC_sendData()* or *USBCDC_receiveData()* returns, even if *size* is small. Thus it operates in the background until completed. One way of describing it is to say that the call to *USBCDC_sendData()*/*USBCDC_receiveData()* starts a chain reaction of USB interrupts that continues until the operation is completed.

This allows the application to proceed freely while the operation is carried out. It has the advantage of not monopolizing program flow during the transfer. However, the software designer must be mindful of the background processing. Any subsequent calls to begin another send/receive operation, as well as any calls that wish to make use of the user buffer identified for that operation, need to first check that no previous operation is underway. That can be done with a call to *USBCDC_intfStatus().*

Preliminary

The background processing of a send operation is shown in Fig. 7. It can be seen that even while the application continues to execute after the return from *USBCDC_sendData(),* the send operation continues and then completes.

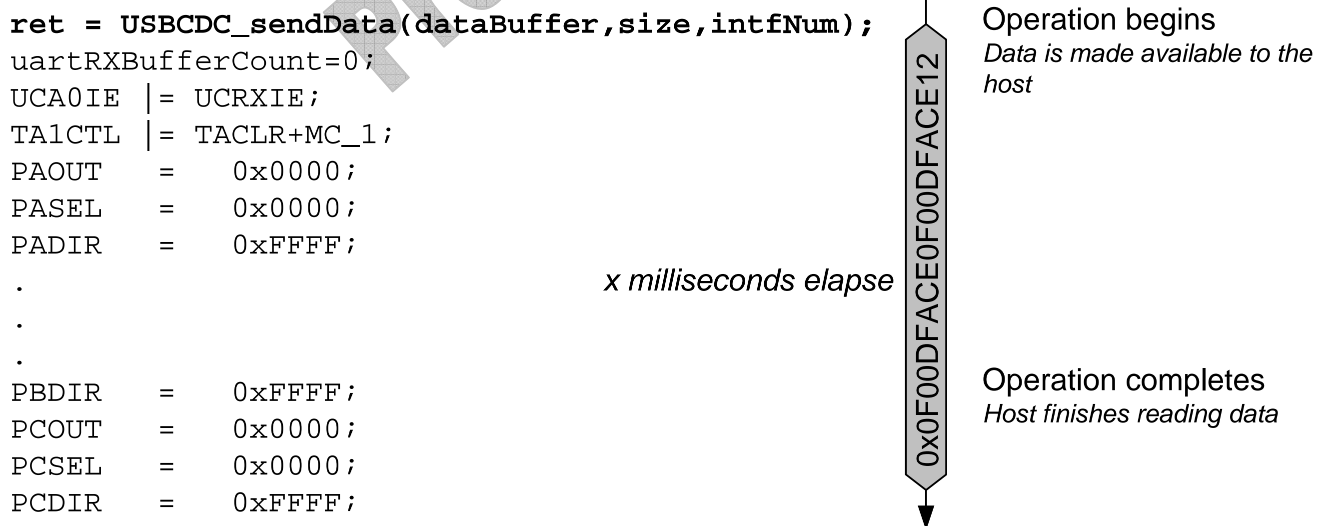


Figure 7. Background Processing in Send Operations

When a send operation is begun, the data is transmitted as quickly as the host, device, and bus conditions will allow. Obviously, the larger the data size, the longer the transmission takes. However, there are additional factors. The transmission in Fig. 7 appears continuous, and in most cases that will be so. But not always; host speed and bus conditions can play a role in how quickly the data is sent, and they can cause the data transmission to be broken into smaller chunks and/or delayed. This is especially true because the CDC uses bulk USB transfers. Bulk transfers significantly aid throughput under the majority of practical conditions, but the tradeoff is the potential for latency that is hard for software to predict. Thus, different latencies may be seen in different bus setups, and they can differ from operation to operation even for the same size of data. This is why it is important for software to be written to account for the potential for variable timelength of open operations.

Receive operations run in the background also, however their dynamics are different. When the USB device application begins a receive operation, the speed and timing at which the data arrives are generally at the host's discretion. Sometimes an application can make predictions of these variables, as in the case of a defined protocol. Sometimes the application has no idea when or how much data will arrive, forcing it to be more open-ended.

Fig. 8 shows a receive operation for 16 bytes over the course of execution, in which the host sends the data in bursts.

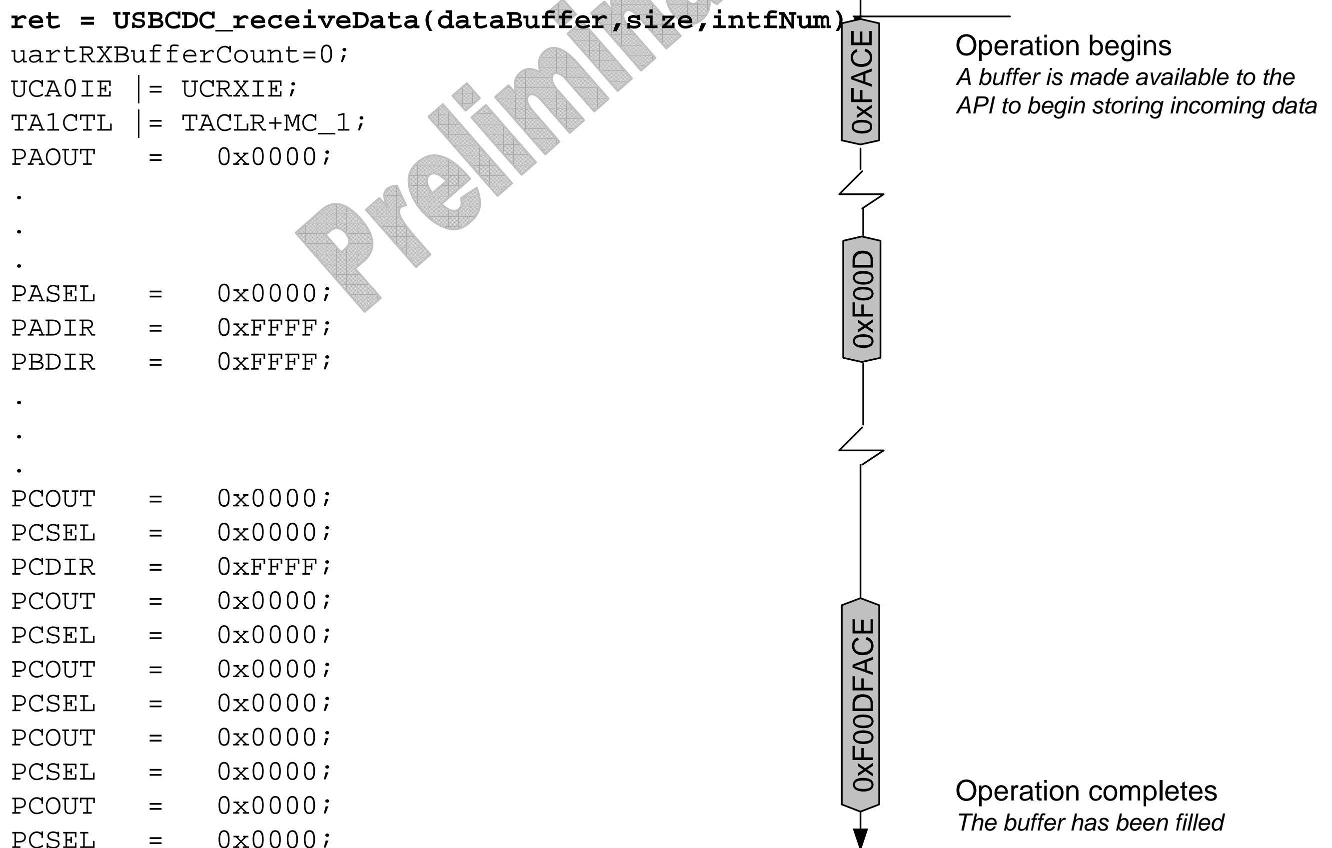


Figure 8. Background Processing in Receive Operations

Preliminary

Fig. 8 shows the operation completing within a contiguous block of code, but that need not be the case. An operation can be started in any part of code and be completed at any point during execution – at the host's discretion, or it can be aborted by the application.

USBCDC_sendData(), USBCDC_receiveData(), and *USBCDC_intfStatus()* have all the return codes necessary to manage this. Also, Sec. 9.3 and 9.4 provides clear, pre-built coding constructs that ensure proper operation.

Preliminary

6 API Function Calls

NOTE: v1.0 of this API stack does not support composite devices. Certain functions have a parameter *intfNum* in them, which is only relevant for composite devices. In v1.0 of the API, this parameter has no effect.

6.1 MSP430 USB Module Management

These calls configure and manage the MSP430's USB module. They are shared among all USB APIs distributed by TI for the MSP430, and they are all defined within the *usb.c* source file.

Table 4. USB Module Management Call Summary

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

Preliminary

6.1.1 BYTE USB_init()

Description

Initializes the USB module by configuring power and clocks, and secures the assocated pins for USB rather than general I/O usage. This should be called prior to any other API functions, perhaps at the beginning of program execution.

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

Parameters

Table 5. Parameters for *USB_init()*

returns	*kUSB_succeed*

6.1.2 BYTE USB_enable(void)

Description

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

This call requires that the clock-related configuration constants be configured properly. These include:

- *USB_PLL_XT:* indicates which oscillator serves as the reference to the PLL
- *USB_XT_FREQ:* indicates the frequency of the crystal on this oscillator

Parameters

Table 6. Parameters for *USB_enable*()

returns	*kUSB_succeed*

Preliminary

6.1.3 BYTE USB_disable(void)

Description

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

This call should be made when VBUS is removed (that is, when *USB_connectionState()* begins to return ST_USB_DISCONNECTED or at a *handleVbusOffEvent()*), after a call to *USB_disconnect(),* in order to avoid unnecessary current draw.

Parameters

Table 7. Parameters for *USB_disable()*

Returns	*kUSB_succeed*

6.1.4 BYTE USB_setEnabledEvents(BYTE events)

Description

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

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

USB_setEnabledEvents() can be thought of in a similar fashion to setting/clearing interrupt enable bits.

See Sec. 7 for more information about events.

Parameters

Table 8. Parameters for *USB_setEnabledEvents*()

word *events*	*kUSB_clockFaultEvent* *kUSB_VbusOnEvent* *kUSB_VbusOffEvent* *kUSB_UsbResetEvent* *kUSB_UsbSuspendEvent* *kUSB_UsbResumeEvent* *kUSBCDC_dataReceivedEvent* *kUSBCDC_sendCompletedEvent* *kUSBCDC_receiveCompletedEvent*
Returns	*kUSB_succeed*

6.1.5 BYTE USB_getEnabledEvents(void)

Description

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

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

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

Parameters

Table 9. Parameters for *USB_getEnabledEvents*()

Returns	*kUSB_clockFaultEvent* *kUSB_VbusOnEvent* *kUSB_VbusOffEvent* *kUSB_UsbResetEvent* *kUSB_UsbSuspendEvent* *kUSB_UsbResumeEvent* *kUSBCDC_dataReceivedEvent* *kUSBCDC_sendCompletedEvent* *kUSBCDC_receiveCompletedEvent*

Preliminary

6.2 USB Connection Management

These calls pertain to the USB connection with the host. They are shared among all USB APIs distributed by TI for the MSP430 and are defined within the *usb.c* source file.

Table 10. USB Connection Management Call Summary

Function	Description
BYTE USB_reset()	Resets the USB module and the internal state of the API.
BYTE USB_connect()	Instructs USB module to make itself available to the PC for connection, by pulling the PUR pin high.
BYTE USB_disconnect()	Forces a disconnect from the PC by pulling the PUR pin low
BYTE USB_forceRemoteWakeup()	Forces a remote wakeup of the USB host
BYTE USB_suspend()	Shuts down the PLL, and optionally the high-frequency crystal.
BYTE USB_resume()	Re-enables the PLL and high-frequency crystal.
BYTE USB_connectionInfo()	Returns low-level information about the USB connection
BYTE USB_connectionState()	Returns the state of the USB connection

Preliminary

6.2.1 BYTE USB_reset()

Description

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

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

Parameters

Table 11. Parameters for *USB_reset* ()

returns	*kUSB_succeed*

6.2.2 BYTE USB_connect()

Description

Instructs the USB module to make itself available to the PC for connection, by pulling the D+ signal high using the PUR pin.

Parameters

Table 12. Parameters for *USB_connect()*

Returns	*kUSB_succeed*

6.2.3 BYTE USB_disconnect(void)

Description

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

Parameters

Table 13. Parameters for *USB_disconnect()*

Returns	*kUSB_succeed*

Preliminary

6.2.4 BYTE USB_suspend(void)

Description

Shuts down the PLL in preparation for a USB suspend state. The PLL is placed into bypass mode, and this causes the USB module to be clocked by the MSP430's VLO oscillator (low-frequency, low-power). The USB module is active. Most USB interrupts are disabled, except for USB resume.

If the configuration constant USB_DISABLE_XT_SUSPEND is non-zero, this function also disables the high-frequency crystal source indicated by USB_PLL_XT. This is the default. Eliminating the oscillator's current draw may be necessary in bus-powered applications in order to meet the USB requirements for suspend current draw.

This function is called automatically within the API in response to a suspend interrupt, and in most cases does not need to be called by the application.

Parameters

Table 14. Parameters for *USB_suspend()*

Returns	*kUSB_succeed*

6.2.5 BYTE USB_resume(void)

Description

Re-enables the PLL, high-frequency crystal, and USB interrupts that were disabled by *USB_suspend().*

This function is called automatically within the API in response to a resume interrupt, and usually does not need to be called by the application.

Parameters

Table 15. Parameters for *USB_resume()*

Returns	*kUSB_succeed*

6.2.6 BYTE USB_forceRemoteWakeup(void)

Description

Prompts a remote wakeup of the USB host. The user must ensure that the configuration constant USB_SUPPORT_REMOTE_WAKEUP reflects that this capability exists, prior to calling this function; otherwise the host will ignore this function.

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

Preliminary

Parameters

Table 16. Parameters for *USB_forceRemoteWakeup()*

Returns	*kUSB_succeed* *kUSB_generalError* *kUSB_notSuspended*

6.2.7 BYTE USB_connectionState(void)

Description

Returns the state of the USB connection, according to the state diagram in Fig. 3.

Parameters

Table 17. Parameters for *USB_connectionState()*

Returns	*ST_USB_DISCONNECTED*

	ST_USB_CONNECTED_NO_ENUM *ST_ENUM_IN_PROGRESS* *ST_ENUM_ACTIVE* *ST_ENUM_SUSPENDED*

6.2.8 BYTE USB_connectionInfo(void)

Description

Returns low-level status information about the USB connection.

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

Parameters

Table 18. Parameters for *USB_connectionInfo()*

Returns	*kUSB_vbusPresent* *kUSB_purHigh* *kUSB_suspended* *kUSB_NotSuspended* *kUSB_enumerated*

Preliminary

6.3 CDC Management and Data Handling

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

Table 19. CDC Management and Data Handling Call Summary

Function	Description
BYTE USBCDC_sendData(const BYTE data, WORD size, BYTE intfNum)*	Begins a send operation to the USB host
BYTE USBCDC_receiveData(BYTE data, WORD size, BYTE intfNum)*	Begins a receive operation from the USB host
BYTE USBCDC_abortSend(WORD size, BYTE intfNum)*	Aborts an active send operation
BYTE USBCDC_abortReceive(WORD size, BYTE intfNum)*	Aborts an active receive operation
BYTE USBCDC_rejectData(BYTE intfNum)	Rejects payload data residing in the USB buffer, for which a receive operation has not yet been initiated
BYTE USBCDC_intfStatus(BYTE intfNum, WORD bytesSent, WORD* bytesReceived)*	Returns status information specific to a particular CDC interface

Preliminary

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

Description

Sends data over CDC interface *intfNum*, of size *size* and starting at address *data.* If *size* is larger than the packet size (as indicated by MAX_PACKET_SIZE), the function handles all packetization and buffer management. *size* has no inherent upper limit.

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

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

See Sec. 5 for a detailed discussion of send operations.

Parameters

Table 20. Parameters for *USBCDC_sendData()*

byte* *data*	An array of data to be sent

word *size*	Number of bytes to be sent, starting from address *data*.
byte *intfNum*	Which data interface the data should be transmitted over
Returns	*kUSBCDC_sendStarted:* a send operation was successfully started *kUSBCDC_intfBusyError:* a previous send operation is underway *kUSBCDC_ busNotAvailable:* the bus is etiher suspended or disconnected *kUSBCDC_generalError:* size was zero, or other error

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

Description

Receives *size* bytes over CDC interface *intfNum,* of size *size,* into memory starting at address *data. size* has no inherent upper limit.

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

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

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

See Sec. 5 for a detailed discussion of receive operations.

Preliminary

Parameters

Table 21. Parameters for *USBCDC_receiveData()*

byte* *data*	An array that will contains the data received.
word *size*	Number of bytes to be received
byte *intfNum*	Which data interface to receive from
Returns	*kUSBCDC_receiveStarted:* A receive operation has been succesfully started. *kUSBCDC_receiveCompleted:* The receive operation is already completed. *kUSBCDC_intfBusyError:* a previous send operation is underway *kUSBCDC_ busNotAvailable:* the bus is etiher suspended or disconnected *kUSBCDC_generalError:* size was zero, or other error

6.3.3 BYTE USBCDC_abortSend(WORD size, BYTE intfNum)*

Description

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

An application may choose to call this function if sending failed due to a surprise removal of the bus, a USB suspend event, or any send operation that extends longer than desired (perhaps due to no open COM port on the host.)

Parameters

Table 22. Parameters for *USBCDC_abortSend()*

word* *size*	Number of bytes that were sent prior to the abort action.
byte *intfNum*	The data interface for which the send should be aborted
Returns	*kUSB_succeed*

6.3.4 BYTE USBCDC_abortReceive(WORD size, BYTE intfNum)*

Preliminary

Description

Aborts an active receive operation on CDC interface *intfNum.* Returns the number of bytes that were received and transferred to the data location established for this receive operation.

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

Parameters

Table 23. Parameters for *USBCDC_abortReceive()*

word* *size*	Number of bytes that were received and are waiting at the assigned address.
byte *intfNum*	The data interface for which the send should be aborted
Returns	*kUSB_succeed*

6.3.5 BYTE USBCDC_rejectData(BYTE intfNum)

Description

This function rejects payload data that has been received from the host for a data interface that does not have an active receive operation underway and is residing in the USB buffer space. When this function is called, the USB receive buffer for the interface is effectively purged, and the data lost. This frees the USB path to resume communication.

Parameters

Table 24. Parameters for *USBCDC_rejectData()*

Returns	*kUSB_succeed*

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

Preliminary

Description

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

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

Parameters

Table 25. Parameters for *USBCDC_intfStatus()*

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

7 Event-Handling

The USB interrupt service routine (ISR) is contained within the API, and is not intended to be modified by the API user. Rather, the API provides *event handling* to the application. Various events are defined, most of which are generated directly from the ISR in response to various interrupt types.

The API calls pre-defined handler functions for these events. These handlers can be thought of in the same way as interrupt service routine handlers. The software designer can write code into these functions without disrupting the ISR. Using the function *USB_setEnabledEvents()* function can be thought of similarly to setting/clearing interrupt enable bits.

Generally speaking, parameters are not passed in or out of the event-handling functions. However, a few of the functions are specific to a particular CDC interface, and in these cases, the interface number is passed into the function.

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

The handler functions are located in *USB_eventhandling.c.* In some cases, the API includes default code within the handlers. This is because this code is considered useful in most applications. However, *USB_eventHandling.c* is considered user space, and the software designer has complete freedom to modify these handlers.

7.1 The Relationship between Interrupts and Events

The event-handling calls are called out of the USB ISR. As such, general interrupts are disabled prior to entering the handler function, and any precautions about good ISR coding apply to event handler functions as well. For examplle, it's recommended that code be kept as short as possible in order to avoid delays in handling other interrupts. (If more functionality is required, the handler can set a flag and then wake the CPU, so that main() can pick up the flag and handle the desired functionality – see Sec. 7.2). The event handler should never enable general interrupts, since unpredictable operation could result, and also since it is always discouraged to enable interrupts from within an ISR.

In fact, for this same reason, *some API function calls should never be made from within the event handlers.* This is because many of them disable and re-enable interrupts. Table 25 shows which functions may and may not be called from the event handlers.

Preliminary

Table 26. Calls that May be Made from Event Handlers

Calls that MAY be called from event handlers	Calls that may NOT be called from event handlers
• Any USB-level call except for USB_init() → USB_xxxxx()	• USB_init() • Any CDC-level call → USBCDC_xxxxxx())

The API calls on the right in Table 25 should only be made from outside the event handlers. Instead of calling from within the handlers, the handlers can set a flag that signals main() to take the required action, then return TRUE to signal main() to wake up (see next section). This is a technique commonly used for interrupt service routines as well.

7.2 Waking from Event Handlers

Events have the option to keep the CPU awake after the ISR (out of which the event handler was called) is completed.

MSP430 ISRs often use the following intrinsic command:

```
__bic_SR_register_on_exit(LPM3_bits);
```

This command modifies the status value on the stack that will get loaded into the status register when the ISR is completed, causing the CPU to resume the main application from a point at which it had entered a low-power mode. (This technique is well-documented in other MSP430 material, including the application note *MSP430 Software Coding Techniques (slaa294).*

The IDEs do not allow this intrinsic to be used inside the event handler functions, only from within the ISR itself. The event handlers are therefore designed to return a boolean value into the ISR that indicates whether the ISR should invoke this intrinsic function to keep the CPU awake. Returning a value of FALSE causes the CPU to go back to sleep after the ISR, while a TRUE value causes the CPU to stay awake.

Preliminary

7.3 Using *USB_setEnabledEvents()*

This function can be used in similar fashion to setting/clearing interrupt enable (IE) bits. Each event tends to occur only in a given USB connection state, and therefore many applications only need to call it once at the beginning of execution.

For those applications that do need to disable an event during execution, the lack of bitwise control within this function adds some difficulty. It is recommended to manage a global BYTE variable that mimics an interrupt-enable register – for example, USB_EVENT_IE – and then pass it into the API using *USB_setEnabledEvents()*. This is shown in Fig. 9.

```
volatile BYTE USB_EVENT_IE;

VOID main(VOID)
{
   WDTCTL = WDTPW + WDTHOLD;
   Init_System();
   USB_init();

   USB_EVENT_IE = kUSB_VbusOnEvent + kUSB_suspendEvent + kUSB_dataReceivedEvent;
   USB_setEnabledEvents(USB_EVENT_IE);
   .
   .
   .
   while(1)
   {
     .
     .
     USB_EVENT_IE &= ~kUSB_dataReceivedEvent;
     USB_setEnabledEvents(USB_EVENT_IE);
   }
```

Figure 9. Managing Event Enables

Preliminary

7.4 Event Handler Functions

A summary of the event handler functions is shown in Table 25.

Table 27. Event Handler Summary

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

Preliminary

7.4.1 byte USB_handleClockEvent(void)

Description

If this function gets executed, it's a sign that the output of the USB PLL has failed. This event will occur even if no USB connection was established, if the event is enabled. This event may have occurred because the high-frequency crystal has failed.

If this event occurs while the USB device is actively connected to the USB host, it probably means the host will soon consider this device non-operational, if it has not already done so by the time this call is executed. As a result, it will most likely suspend the device.

After re-establishing the clock, the application may choose to re-connect to the host from this handler.

7.4.2 byte USB_handleVbusOnEvent(void)

Description

If this function gets executed, it indicates that a valid voltage has just been applied to the VBUS pin. That is, the voltage on VBUS has transitioned from low to high. This generally means that the device has been attached to an active USB host.

The MSP430 hardware and API automatically enable the 3.3V LDO. No action is required by the application in order to enable USB operation; the USB module is enabled by the API prior to the event handler function being executed.

7.4.3 byte USB_handleVbusOffEvent(void)

Description

If this function gets executed, it indicates that a valid voltage has just been removed from the VBUS pin. That is, the voltage on VBUS has transitioned from high to low. This generally means that the device has been removed from an active USB host.

The USB module has likely been powered down, and the connection has been lost. The API responds to this, internally, by disabling the USB module and PLL. It then calls this event handling function, if enabled.

Preliminary

7.4.4 byte USB_handleResetEvent(void)

Description

If this function gets executed, it indicates that the USB host has issued a reset of this USB device. The API handles this event automatically, and no action is required by the application to maintain USB operation. It is provided to the application primarily for informational purposes.

7.4.5 byte USB_handleSuspendEvent(void)

Description

If this function gets executed, it indicates that the USB host has chosen to suspend this USB device. When this occurs, it is important that a bus-powered USB device not consume more than 500uA from VBUS, per the USB 2.0 specification. (Devices not drawing power from VBUS need not worry about the restriction.) The API internally calls *USB_suspend()* in response to being suspended; it then calls this event handler.

The developer of a bus-powered application should be aware of how much current the system is drawing from VBUS. A good use for this handler is to contain code that reduces current draw from VBUS to within the 500uA limit.

7.4.6 byte USB_handleResumeEvent(void)

Description

If this function gets executed, it indicates that the USB host has chosen to resume this USB device from suspend mode. When this occurs, a bus-powered device is no longer restricted to drawing under 500uA from VBUS. The API internally calls *USB_resume()* in response to being resumed; it then calls this event handler.

Upon being resumed, the device is no longer restricted to drawing less than 500uA from VBUS. Therefore, the application may use this event handler to activate functions that were shut down during suspend.

7.4.7 byte USBCDC_handleDataReceived(BYTE intfNum)

Description

This event indicates that data has been received for data interface *intfNum,* but no data receive operation is underway. Effectively, the API doesn't know what to do with this data and is asking for instructions. Either a receive operation can be initiated, or the data can be rejected by calling *USBCDC_rejectData()*. Until one of these is performed, USB transactions cannot continue; any packets received from the USB host will be NAK'ed, and while send operations can be initiated, they will not make any progress in transferring their data to the host.

Preliminary

Therefore, it is critical that this event be handled quickly,. A call to *USBCDC_receiveData()* cannot be made out of this event, due to the restriction of calling API functions from the event handlers, but it may set a flag for main() to begin a receive operation. After this function exits, a call to *USBCDC_intfStatus()* for this CDC interface (but not any other interfaces) will return *kUSBDataWaiting*.

If a receive operation is always begun in advance of data being received from the host, this event will never occur. The software designer generally has a choice of whether to use this event as part of code flow, or to always keep a receive operation open in case data arrives.

7.4.8 byte USBCDC_handleSendCompleted(BYTE intfNum)

Description

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

In applications sending a series of large blocks of data, the designer may wish to use this event to trigger another send operation. A call to *USBCDC_sendData()* cannot be made out of this event, due to the restriction of calling API functions from the event handlers, but it may set a flag for main() to begin a send operation.

7.4.9 byte USBCDC_handleReceiveCompleted(BYTE intfNum)

Description

This event indicates that a receive operation on CDC interface *intfNum* has just been completed. The data can be retrieved. It is located at the address assigned when the receive operation was initiated. If the event occurs, it means that the full size that was requested by the fetch was indeed received.

The designer may wish to use this event to trigger another receive operation. A call to *USBCDC_receiveData()* cannot be made out of this event, due to the restriction of calling API functions from the event handlers, but it may set a flag for main() to begin a receive operation.

Preliminary

8 Configuration Constants

A collection of constants are located in *descriptors.h* that control operation of the API. These are implemented as constants rather than function parameters, because their values are considered to be compile-time values that do not change during operation.

The constants associated with basic USB-level functionality are shown in Table 28. Those that are specific to the CDC functionality are shown in Table 29.

Table 28. USB-Specific Configuration Constants

Constant	Description
USB_VID	Four-digit hex value reflecting the USB vendor ID, as assigned by the USB-IF.
USB_PID	Four-digit hex value reflecting this device's product ID, as assigned by the OEM.
USB_STR_INDEX_SERNUM	Indicates the index of the string descriptor that contains the serial number string. A serial number plays an important role in helping the OS identify this individual device, and helps it associate specific settings with this device. A specific combination of VID, PID, and serial number should be unique to any USB device in existence. If this value is non-zero, then a string descriptor must be located at that index. If no serial number is desired, this index should be assigned a value of 0.
USB_SUPPORT_REM_WAKE	Controls whether the remote wakeup feature is supported by this device. A value of 0x20 indicates that it is supported (this value is inserted into the *bmAttributes* field in the configuration descriptor). A value of zero indicates remote wakeup is not supported. Other values are undefined, as they will interfere with *bmAttributes.*
USB_SUPPORT_SELF_POWERED	If the device is self-powered to any degree, this constant should be set to 0x40. If it is fully bus-powered while attached to the host, it should be set to 0x00. By default, it is set to 0x40.
USB_MAX_POWER	This is the value to be reported in the *bMaxPower* field of the configuration descriptor. The maximum amount of current that the device will draw from the host, in mA, is the value of *USB_MAX_POWER* times two. (a value of 50 = 100mA).
USB_NUM_CONFIGURATIONS	The current version of the API restricts this value to 1.
USB_DMA_TX	Selects the DMA channel to be used for transmit operations. Must reflect a channel that is implemented in the chosen MSP430 derivative, and must be different than the channel used for receive operations. 0x00 for ch. 0, 0x01 for ch. 1, up to 0x07 for ch. 7. If no DMA channel is to be used, rather relying on CPU transfers, set this to 0xFF.
USB_DMA_RX	Selects the DMA channel to be used for receive operations. Must reflect a channel that is implemented in the chosen MSP430 derivative, and must be different than the channel used for transmit operations. 0x00 for ch. 0, 0x01 for ch. 1, up to 0x07 for ch. 7 If no DMA channel is to be used, rather relying on CPU transfers, set this to 0xFF.
USB_PLL_XT	This selects the high-speed oscillator used to source the USB PLL. It must have a value of either 1 (for XT1) or 2 (for XT2).
USB_DISABLE_XT_SUSPEND	If non-zero, then USB_suspend will disable the oscillator indicated by USB_PLL_XT. If zero, then USB_suspend will not affect the oscillator. USB_resume() will always restore the oscillator in either case.
USB_MCLK_FREQ	The MCLK frequency while USB is operating, in Hz. (12500000 = 12.5MHz) The API uses this only to determine delays during startup of the USB module; it does not configure MCLK. This must be done outside the API by the application. Since it is used to determine delays, then if MCLK speeds are changed during operation, this constant should be set to the highest frequency that is ever in use while USB is active.

Preliminary

VER_FW_H / VER_FW_L	The high and low bytes of the device release number, which is reported in the *bcdDevice* field of the device descriptor. As the name implies, the values are binary-coded decimal.

Table 29. CDC-Specific Configuration Constants

Constant	Description
MAX_PACKET_SIZE	This is the size of packets that are used to exchange serial data over USB. Larger packets are generally more efficient when transceiving large chunks of data. In most CDC situations, this value should be 64, the largest allowed by the USB specification for a bulk endpoint.

Preliminary

9 Practical Matters: How to Write USB Programs with the API

Sec. 4-5 gave conceptual information about how the API works, and Sec. 6-8 serve as a reference for function calls. This section builds on the others by discussing the practical considerations of writing a USB program with the API. Please use the prior sections as a reference.

9.1 Considerations When Designing a USB Program

The considerations in this section are implemented into the constructs of Sec. 9.2, 9.3., and 9.4. So although Sec. 9.1 should be required reading for any USB program, the reader may rest assured that the constructs and application examples implement what's required.

9.1.1 Robustness: Handling Surprise Removal or Suspend

While developing code, it's important for the system designer to consider events that may occur unexpectedly. Due to the nature of USB – and somewhat unlike an RS232 connection -- it usually can't be assumed the bus is always available, or that it only becomes unavailable at certain times. At any time, the bus may become suspended by the host, or disconnected by the user (a "surprise removal").

A device should always recover from these events gracefully. Software must be built with the assumption that the bus could disappear at any time. Code should not execute under an errant assumption that the bus is available when it is not. It is similar to considering how interrupts may occur at any time within any embedded application; the software designer must ensure that conflicts don't disrupt execution.

Preliminary

9.1.2 How Changes in USB State Influence the Code Flow

As discussed in Sec. 4 a USB device must anticipate being in several different states, returned by *USB_connectionState()*. The ones in which a device will spend most of its time are *disconnected*, *enumerated-active*, and *enumerated-suspend*.

For most USB devices, these states have a deep impact on functionality, and therefore change the nature of its operation at any given time. For example, when not connected to a host, a digital still camera's function is to take pictures. Once connected to the host, its function is generally to upload pictures, and it might even prevent the user from taking pictures. Thus the state of the bus has completely altered the software flow. So, one of the first decisions the designer should consider is how functionality will change (or if it will change) according to bus state.

The states change according to external events, which means they can change at any time. Implementing a main loop that periodically calls *USB_connectionState()* gives the main loop a chance to respond to them.

9.1.3 Using Events to Refresh the Main Loop

If a low-power mode is invoked in the loop, it's often important that the CPU be awakened in response to a state change, so that it may "update" the main loop. This can be done by enabling the following event handlers and making sure they return TRUE:

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

Each of these events correlate with a change in state. Waking the CPU gives the program a chance to respond to the state change. This is somewhat application-specific, however; if no pertinent code exists within the main loop, then it may not be necessary.

See Sec. 7.2 for further discussion on using the event handlers to wake the main loop.

9.1.4 Changing Power Circuitry in Response to Suspend/Resume

If a USB device is drawing any power from the host, and it becomes suspended by the host, it's required to reduce its draw over VBUS to <500uA, within 10ms of the suspend event. Depending on the way power is provided on the device, software may need to take action, and do so quickly. Additionally, functionality might need to be reduced in order to meet the suspend limitation. All of this can have an effect on software flow.

Preliminary

9.2 Main Loop Framework

A recommended main loop construct is shown in Fig. 10.

```
VOID main(VOID)
{
    WDTCTL = WDTPW + WDTHOLD;                  // Configure watchdog
    Init_System();
    USB_init();
    USB_setEnabledEvents(kUSB_VbusOnEvent + kUSB_VbusOffEvent +
      handleSuspendEvent + handleResumeEvent);

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

    while(1)
    {
        switch(USB_connectionState())
        {
           case ST_USB_DISCONNECTED:
                __bis_SR_register(LPM3_bits + GIE);          // Enter LPM3
                function_as_bus_disconnected();
                break;

           case ST_USB_CONNECTED_NO_ENUM
                break;

           case ST_ENUM_IN_PROGRESS:
                break;

           case ST_ENUM_ACTIVE:
                __bis_SR_register(LPM0_bits + GIE);          // Enter LPM0
                function_as_bus_active();
                break;

           case ST_ENUM_SUSPENDED:
                __bis_SR_register(LPM3_bits + GIE);          // Enter LPM3
                function_as_bus_suspended();
                break;

           case ST_ERROR:
                _NOP();
                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
```

Preliminary

```
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
```

Figure 10. Main Loop Framework

Software flow within the main loop "forks" depending on the state of the USB, creating several "smaller" main loops. This is as discussed in Sec. 9.1.2.

This framework causes the device to attempt a connection to the host when the it's attached, since this behavior is fairly standard to the USB experience. This code is in the *handleVbusOnEvent()* handler, which gets called automatically at a VBUS-on transition. However, there is no rule that requires this behavior, and so it could be removed if desired.

handleVbusOnEvent() can also be called at the top of main(). Normally the application doesn't call the event handler functions; the API does this. However, this is an exception. This is because the event handler only gets called when VBUS transitions, and if VBUS is already on, there is no transition. Thus it must be done "manually".

In this framework, any event handler that reflects a state change is enabled and returns TRUE. This causes the main loop to awaken from a low-power mode, if one was entered, to allow it to "refresh" itself for the new state. This is as discussed in Sec. 9.1.3.

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

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

Preliminary

9.3 Recommended Constructs for Send Operations

Two things need to be considered with send operations:

- *Background Processing*. A successfully-started send operation may still be executing after the call to *USBCDC_sendData()* returns, operating in the background. Therefore a subsequent call to *USBCDC_sendData()* might find that the interface is already busy. This was discussed in more detail in Sec. 5.4.
- *Bus disconnect/suspend*. The end user may disconnect the bus at any time, or the host may suspend the device at any time. This may cause a call to *USBCDC_sendData()* to stall. Either event could cause a disruption to code flow, if it wasn't written to anticipate these events.

The API functions return all the data needed to handle these events. Code must be written to anticipate all the return values, which enables it to account for these factors.

The following coding construct functions handle these for the software designer. Further, the API provides them in the file *usbcdc_constructs.c,* and the software designer is encouraged to use them.

Preliminary

9.3.1 Background Processing

As discussed in Sec. 5.4, the value of send/receive operations is that they prevent the USB code from monopolizing code flow, as they would otherwise. Effectively it "uncouples" the device's application from the host so that a delay on the host and/or bus doesn't result in a locked-up device. It becomes increasingly valuable with larger operation sizes, busy buses, and slow hosts.

Because send operations run in the background, the code shouldn't make assumptions about when it has completed. Similarly, it shouldn't make assumptions about the return code of a call to *USBCDC_sendData()* or *USBCDC_receiveData()*. These return codes indicate if the call failed because an operation was already underway. (The application can also determine if the operation has been completed by making a call to *USBCDC_intfStatus()*.)

Generally speaking, there are two approaches to ensuring a send operation is complete before attempting another. They are shown in Table 30.

Table 30. Approaching to Sending Data

Technique	Description	Advantage	Disadvantage
Post-call polling	Execute a polling loop *after* the call to *USBCDC_sendData(),* which waits until the operation is complete.	Afterward, the software designer can know for certain that no send operation is underway.	No other code can execute while the polling loop is occurring; thus, it's more inefficient.
Pre-call polling	Allow send operation to run in the background, and then poll for an open operation *prior* to any subsequent call to *USBCDC_sendData()*.	Takes advantage of background processing, allowing the transfer to happen while other code executes; always more efficient	User buffer for this operation is not free for editing, since the operation may still be in progress

Some engineers may find post-call polling more intuitive. It can usually be used without consequence in small operations, since they usually complete quickly and therefore don't affect efficiency.

However, pre-call polling is always more efficient to some degree or another, since it gives the operation more time to complete in the background and is less affected by busy bus conditions. It is almost always recommended over post-call polling for this reason.

Whether post-call or pre-call polling is used, it's important they be used consistently. In most cases, the two methods should not be mixed within the same program – the consequences would either be pointlessness (a post-call poll followed by a pre-call poll) or defeating the purpose (the other way around).

Examples later in this section illustrate these approaches.

9.3.2 Anticipating a Lost Bus

The USB connection might be disconnected or suspended at any time. As discussed in Sec. 9.2, this has an impact on high-level code flow. However, it's also usually important that code *within* the main loop doesn't "hang up" or execute incorrectly because it's operating under the assumption that the bus is still connected, when it's not. It's usually better that execution break from the main loop when this condition has been detected.

Preliminary

Generally speaking, this means writing code that is aware that the following functions can return *kUSBCDC_busNotAvailable:*

- *USBCDC_sendData()*
- *USBCDC_receiveData()*
- *USBCDC_intfStatus()*

When this return value is encountered, some cleanup functions might be necessary, and then the main loop should quickly complete its run (or break out of it) so that the next call to *USB_connectionState()* can be made, allowing code flow to reflect the new conditions.

9.3.3 Example Send Constructs

The following code constructs completely implement the aforementioned concepts. They're also provided in main.c of the API code files as functions, and the reader is encouraged to use them in applications.

Note that none of these functions check for a return value of *kUSBCDC_generalError*. This return value only happens if the size parameter is zero; it is simply assumed the application won't make that mistake. If this condition can occur, be sure to check for it prior to calling these functions.

9.3.3.1 Post-Call Polling: *sendData_waitTilDone()*

A robust post-call polling function is shown in Fig. 11.

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

  ret = USBCDC_sendData(dataBuf,size,intfNum);  // ret sendStarted or busNotAvailable
  if(ret == kUSBCDC_busNotAvailable)
    return 2;

  while(1)                                     // It was sendStarted
  {
    ret = USBCDC_intfStatus(intfNum,&bytesSent,&bytesReceived);
    if(ret & kUSBCDC_busNotAvailable)          // This may happen at any time
      return 2;
    else if(ret & kUSBCDC_waitingForSend)
    {
      if(ulTimeout && (sendCounter++ >= ulTimeout))
        return 1;
    }
    else return 0;                             // It succeeded!
  }
}
```

Figure 11. Robust Post-Call Polling

Preliminary

Notice this function checks every return value from *USBCDC_sendData().* It can then return TRUE or FALSE, which determines whether or not the calling function should exit the main loop. This would occur if the bus was found to not be available, or if the function timed out.

The main purpose of the timeout function is to avoid being caught here infinitely. One of the more likely explanations for the function timing out is that the COM port has not been opened on the host. So, even though the USB connection is active, the host isn't retrieving data from the device's USB buffers, causing execution to poll *USBCDC_intfStatus()* indefinitely – until it times out. The timeout function takes into account MCLK speed.

Some applications that use the watchdog timer might choose to rely on the watchdog to cover this timeout functionality. This would of course involve careful placement of watchdog resets, and accepting that code execution will reset if this event occurs.

9.3.3.2 Pre-Call Polling: *sendData_inBackground()*

Fig. 12 shows a robust pre-call approach.

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

  while(USBCDC_intfStatus(intfNum,&bytesSent,&bytesReceived) & kUSBCDC_waitingForSend);
  {
    if(ulTimeout && ((sendCounter++)> ulTimeout))
      return 1;
  }

  // By now, return from sendData( ) must be either busNotAvailable or sendStarted
  if(USBCDC_sendData(dataBuf,size,intfNum) == kUSBCDC_busNotAvailable)
    return 2;
  else return 0;                                    // It succeeded!
}
```

Figure 12. Robust Pre-Call Polling

As with the post-call method in Fig. 11, a return code of FALSE indicates the call failed, for reasons that should cause the calling function to consider exiting quickly from the main loop. And as with the pre-call method, a timeout function may be employed.

Preliminary

9.3.3.3 Using the Send Constructs

Notice the names of the functions above: *sendData_waitTilDone()* and *sendData_inBackground()*. These names imply their strengths.

The post-call method guarantees the operation to be completed when the call returns, and because of this, the user buffer (*dataBuf*) could be edited in the very next instruction without harming operation. This would not be possible with the pre-call method in Fig. 12, but this method is more efficient, because it allows the send operation to run in the background until the next time this function is called. No changes should be made to *dataBuf* between the calls, since it may still be in use for the send operation.

As indicated earlier, *sendData_inBackground()* is always recommended. *sendData_waitTilDone()* is provided for those engineers wishing to avoid thinking about background operation.

Figures 13 and 14 show the post-call and pre-call functions in use.

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

No concurrency; operations complete after return from sendData_waitTilDone()

Preliminary

Figure 13. Post-Call Function Usage: *sendData_waitTilDone()*

Figure 14. Pre-Call Functon Usage: *sendData_inBackground()*

Preliminary

In the post-call polling example, the operation is fully complete when *sendData_waitTilDone()* returns. In the pre-call polling example, the first operation might finish in time for the second call to *sendData_inBackground()*, or *sendData_ inBackground()* may find it needs to poll until the first operation is done. Then, the second call to *sendData_ inBackground()* may result in an operation that stays active even while the main loop completes and comes back to the first call to *sendData_ inBackground()*.

For this reason, note a slight difference in how the user buffers are utilized: the post-call method is allowed to edit the original buffer at any point after the original send function. But the pre-call method does not attempt this; instead, it uses an X/Y buffer scheme. This is because when using the pre-call method, the application should not edit the buffer until it can be guaranteed the operation is complete; and the only way to do this is to execute another call to *sendData_ inBackground()* (Or, execute some other code that makes use of *USBCDC_intfStatus()*.) Of course, when this second call happens, another send operation is begun, but the first buffer can now be modified because its send operation is guaranteed to be complete. Because of the efficiency of background operations, this X/Y scheme is the way to achieve maximum throughput under all bus/host/application conditions.

Both functions break from the main loop if the bus fails. Most often, the application should abort the send operation in this case, because a later resumption of the bus would allow the operation to resume mid-stream, and the host may not know how to interpret it in this later context. However, this is application-specific, which is why the operation isn't automatically aborted within the API.

9.4 Recommended Constructs for Receive Operations

Receive constructs can be quite different from send constructs. Whereas a send operation completes as quickly as the host, device, and bus allow the data to be transmitted, a receive operation has the added variable of when the host is ready to send, and this may depend on the end user – a real-time variable. Attempting to key the program execution too tightly to the arrival of data can result in unwanted coupling of the device's application to the host's timing. In other words, the device's software could lock up while waiting for the host. The background processing of the API's receive operation naturally lends itself to solve this problem, but the software designer still needs to decide when to start an operation and how large to make it.

There also might be a difficulty in knowing how many bytes the host is going to transmit. Some applications know this, and some don't. If it's not known, it can be difficult for the system designer to know what size to use for the receive operation.

Preliminary

Together, these two factors create a wider range of possible constructs than with send operations. The constructs that follow are common ones, and they cover most usage scenarios. A deep understanding of receive operations may open up even more possibilities.

Here's a summary of the suggested receive constructs:

- *Reactive Receive*. A receive operation is opened which is no larger than the amount of data already in the USB buffers. This is useful when the size of an overall operation is unknown, and the application wishes to "build" the data one portion at a time.
- *Fixed-Size Receive*. A simple receive operation is opened, with a size that is known. The receive operates in the background, and the application is alerted with *handleReceiveCompleted()* when it is done.
- *Continuously-Open Receive.* The application maintains a large receive operation open at all times. When it wants to use the data, it aborts the receive, extracts the data, and opens a new receive operation. This approach is useful when the size is unknown, and when the software designer wishes to impose the least dependency on the host's timing.
- *Wait-Until-Done Receive.* This construct bears the most similarity to the send constructs: a receive operation is opened, and the application polls until the operation is complete. It is highly dependent on the host sending the expected data, and the size must be known with certainty.

9.4.1 Background Processing

As with send operations, receive operations operate in the background. This characteristic is even more valuable for receive operations, since not being so would have forced an application to either poll frequently for any new data that has arrived, or hold its execution in one place while the data (hopefully) is received. The latter approach could put the application's execution at the mercy of the host, potentially locking it up for an amount of time noticeable to the end user. Background operation prevents this.

Unlike with send operations, an application is generally very aware if a receive operation is underway. After all, it's paying attention to when the data arrives, because it plans to do something with it when it does. (This is in contrast to sending, when the application's attention tends to leave the send operation after the call to *USBCDC_sendData()* is made.)

9.4.2 Example Receive Constructs

The following sections discuss different approaches to receive operations. Each is best suited for different tasks. Unlike the send constructs, most of them are not well-suited to wrapping within a function call, for reasons described above.

9.4.2.1 Reactive Receive: Establishing Operations Only for Data Already in the USB Buffers

Preliminary

If the operation's size is no more than the amount waiting in the USB buffer, the operation has no timing dependency on the host. It's simply a matter of copying data from the USB buffer to the user buffer, with no further bus activity, and so it usually happens immediately with the call to *USBCDC_receiveData()*. This approach creates a great deal of flexibility, and allows a construct like the one shown in Fig. 15.

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

  while(bytesInBuf = USBCDC_bytesInUSBBuffer(intfNum)) // # of bytes already in USB buffer
  {
    if((WORD)(currentPos-dataBuf+bytesInBuf) > size)   // Break if limit has been reached
      break;

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

Figure 15. Reactive Receive

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 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, because data could have arrived while the first data was being retrieved. This situation could repeat indefinitely, so it continues to poll until *USBCDC_bytesInUSBBuffer()* returns zero. This will happen when the stream of data from the host stops.

This method works especially well the timing and size of the incoming data are unknown. The application can simply "build" the received data one portion at a time as the portions arrive. The best example of this is in the case of text entry from a terminal application on the host, in which the text is delimited by pressing the "return" key. The number of incoming bytes isn't known, and neither is the timing. This method can be used to receive the string one portion at a time. This is demonstrated in several of the application examples that accompany the API.

9.4.2.2 Fixed-Size Receive: Simply Opening an Operation

Any time the size of an operation is known, the matter can be as simple as opening a receive operation of that size, and using *handleReceiveCompleted()* to trigger the code that processes the received data. Such an operation could be started at any time.

Preliminary

```
if(USBCDC_receiveData(dataBuf,size,intfNum) == kUSBCDC_busNotAvailable)
  break;
```

Figure 16. Fixed-Size Receive

Any return value other than *kUSBCDC_busNotAvailable* would indicate a successful receive operation start. Therefore it's the only one trapped.

This method can be very useful in command/status protocols of the software designer's making. For example, consider a simple command packet of this format:

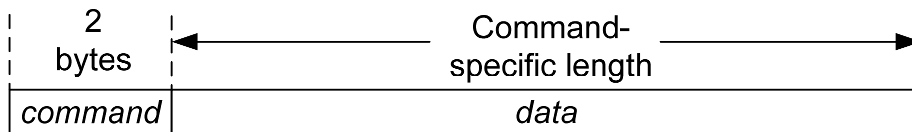


Figure 17. Command Packet

An application could implement this as two separate fixed-receive constructs. The first one has a size of two bytes, to receive the command. When the command is identified, and its corresponding data length known, a second receive operation could be established for that length. *handleReceiveCompleted()* would be used in both cases to flag when the data is received.

9.4.2.3 Continuously-Open Receive: Maintaining a Large Operation Open at All Times

This approach is akin to keeping an “open ear” at all times. In is the least structured approach, and the most de-coupled from the host in terms of timing. It makes no assumption about when data is arriving and little assumption about how much data there will be.

It consists of maintaining a larger-than-needed open receive operation at all times, one which never completes. This way, data always has a place to go. Then, at a timing that is convenient to the application, it can abort the operation and use the data that’s been received. It can even use the data received so far without aborting the operation. Soon after aborting, it should re-open another large operation, to maintain the continuously-open state.

This approach is shown in Fig. 18.

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

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

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

Preliminary

Figure 18. Continuously-Open Receive

The main loop in this example will execute continuously, always checking to make sure there’s an open receive operation, and – separately – checking if received data has reached a certain threshold and handling it. It’s important to note that the data handling is completely isolated from the code that starts the receive; they aren’t linked in terms of timing, except that the code to open a subsequent receive operation should run soon after the preceding one is aborted. .

This approach isn't event-driven in any respect; it's purely a polling approach. For this reason, the main loop must either always be active at all times, or it must be awakened by some other kind of periodic event; perhaps a timer. Otherwise, the main loop won't be able to poll for received data.

If such a "heartbeat" (timer interrupt) already exists within the system, this approach could be preferred even if another method described in this section works as well. This is because it might reduce competition between the timing driven by the host and the timer-driven heartbeat of the application.

9.4.2.4 Wait-Until-Done (Post-Call Polling)

The continuously-open receive method is the most flexible in terms of timing; this one is the most inflexible. It commits itself to the idea that the host is definitely going to send a certain number of bytes within a short period of time. Its construction is similar to the post-call polling method described for send operations: it calls the receive operation and then polls until done. It's shown in Fig. 19.

```
// Assumes a receive operation is NOT underway; also assumes size is non-zero.
// Returns 0 if bus disappeared, 1 if successful, and 2 if it just timed out
BYTE receiveData_waitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)
{
  ULONG rcvCounter = 0;
  BYTE ret;
  WORD bytesSent, bytesReceived;

  ret = USBCDC_receiveData(dataBuf,size,intfNum);
  if(ret == kUSBCDC_busNotAvailable)
    return 2;                                   // Indicate bus is gone
  if(ret == kUSBCDC_receiveCompleted)
    return 0;                                   // Indicate success

  while(1)                                      // ret was receiveStarted
  {
    ret = USBCDC_intfStatus(intfNum,&bytesSent,&bytesReceived);
    if(ret & kUSBCDC_busNotAvailable)           // This may happen at any time
      return 2;
    else if(ret & kUSBCDC_waitingForReceive)
    {
     if(ulTimeout && (rcvCounter++ >= ulTimeout))
       return 1;                        // Indicate no success, but just timed out
    }
    else return 0;                      // Indicate success
  }
}
```

Preliminary

Figure 19. Robust Post-Call Polling Receive

If the code's assumption that data will soon be arriving is wrong, it returns a value of 2. The timeout feature on this function may need adjusting by the software designer to match the expectations of the host's timing. It the timeout isn't used, and the host doesn't send data, it will lock up forever or be reset by the watchdog.

It's worthwhile to compare this method to the fixed-size receive method. The two are the same, in that in each case the exact size is known, and in both cases there is some timing dependency on the host. The difference is that in this case the application's code execution is completely tied to the operation completing successfully; in the other case, execution runs in the background and can recover more easily from an unexpected failure of the host.

9.5 Send/Receive Construct Functions: Summary

This section serves as a reference for the functions described in Sec. 9.3 and 9.4. The functions are found in the file *usbcdc_constructs.c.*

9.5.1 BYTE sendData_waitTilDone(BYTE dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)*

Description

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

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

In many applications, the return value can simply be evaluated as true or false, where "true" means that the call failed, and the application may choose to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

Parameters

Preliminary

Table 31. Parameters for *sendData_waitTilDone* ()

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

9.5.2 BYTE sendData_inBackground(BYTE dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)*

Description

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

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

In many applications, the return value can simply be evaluated as true or false, where "true" means that the call failed, and the application may choose to break from execution. Other applications may wish to handle return values 1 and 2 in different ways.

Parameters

Table 32. Parameters for *sendData_inBackground*()

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

Preliminary

9.5.3 WORD receiveDataInBuffer(BYTE* dataBuf, WORD size, BYTE intfNum)

Description

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

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

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

Parameters

Table 33. Parameters for *receiveDataInBuffer*()

Returns	The number of bytes received into *dataBuf*

9.5.4 WORD receiveData_waitTilDone(BYTE* dataBuf, WORD size, BYTE intfNum, ULONG ulTimeout)

Description

Receives *size* bytes into *dataBuf,* over interface *intfNum.* The function doesn't return until all *size* bytes have been received, or until the timeout is reached (if *bTimeout* is TRUE). The function assumes that size is non-zero.

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

As discussed in Sec. 9.4, there are other approaches to receiving data that are usually better than this one. See that section for alternatives.

Parameters

Table 34. Parameters for *receiveData_waitTilDone*()

Returns	*0:* the call succeeded; all data has been sent *1:* the call timed out 2: the bus is unavailable.

Preliminary

10 References

- MSP430 5xx User's Guide (http://www.ti.com/msp430)
- USB 2.0 specification (http://www.usb.org/developers/docs/)
- Communications Device Class specification and PSTN Subclass specification (http://www.usb.org/developers/devclass_docs#approved)
- Microsoft MSDN website (http://www.microsoft.com/whdc/connect/usb/usbfaq_intro.mspx)
- *MSP430 Software Coding Techniques (slaa294)*

Preliminary