USB Device Datasheet USB V 1.90

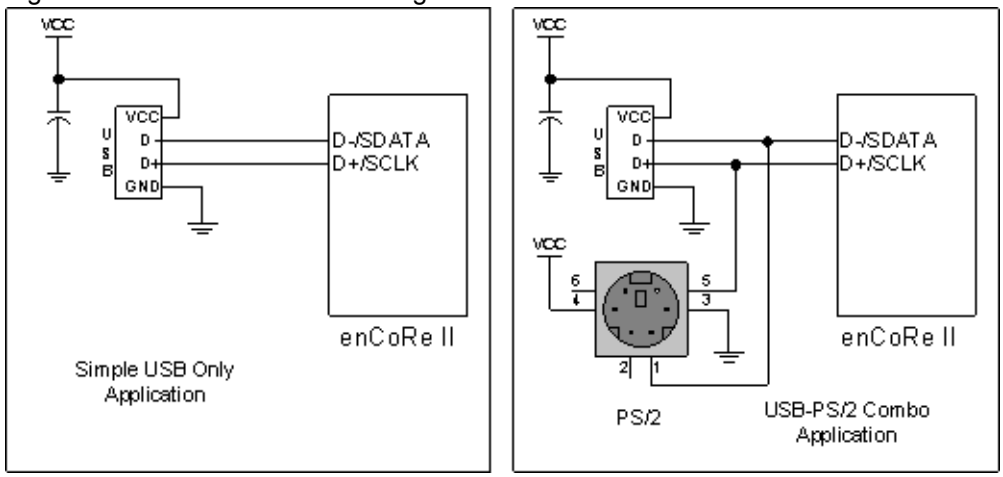
Copyright © 2004-2014 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	API Memory (Bytes)		Pins
	Flash	RAM	
CY7C639/638/633xx, CYRF69xx3			
	1499	41	2
Optional HID Class Driver	+461	+27	2
String Descriptor	~40 + 9 per string descriptor	0	2

Features and Overview

- USB device interface driver
- Support for interrupt and control transfer types
- Setup wizard for easy and accurate descriptor generation
- Runtime support for descriptor set selection
- Optional USB string descriptors
- Optional HID class support
- Optional PS/2 support for USB-PS/2 combination devices (Note: PS/2 is not supported on CYRF69xx3 devices)

Figure 1. USB Device Block Diagrams



Functional Description

The USB Device User Module provides a USB Chapter 9 compliant device framework. The user module gives a low level driver for the control endpoint that decodes and dispatches requests from the USB host. The user module supports the HID Class. USB descriptors can be configured with the USB Setup Wizard.

Timing

The USB Device User Module supports USB 2.0 low speed operation on the enCoRe™ II device family.

Placement

The USB User Module occupies the USBXCVR and USBSIE blocks. Alternate placement is not available.

Parameters and Resources

The USB Device User Module does not use the PSoC Designer User Module Parameter Grid Display for personalization. Instead, it uses a form driven USB Setup Wizard to define the USB descriptors for the application. From the descriptors, the wizard personalizes the user module.

Application Programming Interface

The Application Programming Interface (API) routines in this section allow programming control of the USB User Module. The following sections describe descriptor generation and integration and list the basic and device specific API functions. Developers need a basic understanding of the USB protocol and familiarity with the USB 2.0 specification, especially Chapter 9, USB Device Framework.

In the API descriptions, parameter references for bDevice, bInterface, and bEP refer to the device index, interface index, and endpoint number. The values are derived implicitly from the USB Setup Wizard.

bDevice ranges from 0 to the number of device descriptors defined in the USB Setup Wizard and are positionally dependent to the order in the USB Setup Wizard (bDevice = 0 is the top entry, bDevice = 1 for the second entry and so on).

bInterface ranges from 0 to the number of interfaces defined for a configuration and is positionally dependent to the order in the USB Setup Wizard.

bEP is the endpoint number specified in the Endpoint Descriptor.

Note The API routines for the USB user modules are not re-entrant. Because they depend on internal global variables in RAM, executing these routines from an interrupt is not supported by the API support supplied with this user module. If this is a requirement for a design, contact the local Cypress Field Application Engineer.

Table 1. Basic USB Device API

Function	Description
void USB_Start(BYTE bDevice)	Enables user module for using the data Device
void USB_Stop(void)	Disable user module.
BYTE USB_bCheckActivity(void)	Checks and clears the USB Bus Activity Flag. Returns 1 if the USB was active since the last check, otherwise returns 0.
BYTE USB_bGetConfiguration(void)	Returns the currently assigned configuration. Returns 0 if the device is not configured.

Function	Description
BYTE USB_bGetEPState(BYTE bEP)	Returns the current state of the specified USB endpoint. 2 = NO_EVENT_ALLOWED 1 = EVENT_PENDING 0 = NO_EVENT_PENDING When an IN endpoint is configured during USB enumeration, the endpoint mode is set to NAK IN tokens and the endpoint state is set to EVENT_PENDING. EVENT_PENDING is equated with IN_BUFFER_EMPTY indicating that the endpoint is ready to accept new data for transfer to the host. The application loads the endpoint when the endpoint state is IN_BUFFER_EMPTY. The USB_XLoadEP function updates the endpoint mode to ACK the next IN token and sets the endpoint state to IN_BUFFER_FULL. When an IN transfer is ACK'ed, the SIE automatically changes the endpoint mode to NAK subsequent transfers. The endpoint ISR updates the endpoint state to IN_BUFFER_EMPTY. When an OUT endpoint is configured during USB enumeration, the endpoint mode is set to NAK OUT tokens and the endpoint state is set to NO_EVENT_PENDING. NO_EVENT_PENDING is equated with OUT_BUFFER_EMPTY indicating that the endpoint is ready to accept new data from the host. The application provides data flow control by enabling the OUT endpoint using the USB_EnableEP function. After the OUT transfer is complete, the endpoint ISR sets the endpoint state to OUT_BUFFER_EMPTY.
BYTE USB_bGetEPAckState(BYTE bEPNumber)	Identifies whether ACK was set by returning a non-zerovalue.
BYTE USB_bGetEPCount(BYTE bEP)	Returns the current byte count from the specified USB endpoint.
BYTE USB_bRWUEnabled(void)	Returns 1 if the REMOTE WAKEUP has been enabled by the host, otherwise returns 0.
void USB_XLoadEP(BYTE*pSrc)	Loads and enables the specified USB endpoint for an IN transfer. **Note** USB_XLoadEP should not be call directly. Rather, the macro USB_LoadEP should be used.
C Macro USB_LoadEP(BYTE bEP, BYTE *pSrc, BYTE count)	Loads and enables the specified USB endpoint for an IN transfer.
BYTE USB_bReadOutEP(BYTE bEPNumber, BYTE*pData, BYTE bLength)	Reads the specified number of bytes from the EndpointRAM and places it in the RAM array pointed to by pData.The function returns the number of bytes sent by the host.
void USB_EnableEP(BYTE bEP)	Enables the specified USB endpoint. This function is used for OUT endpoints to set the endpoint mode to ACK the next OUT transfer. It also sets the endpoint state to OUT_BUFFER_EMPTY.
void USB_DisableEP(BYTE bEP)	Disables the specified USB endpoint. This function is used for OUT endpoints to set the endpoint mode to NAK subsequent OUT transfers. It does not change the endpoint state.

Function	Description
void USBFS_SetPowerStatus(BYTE bPowerStatus)	Sets the device to self powered or bus powered.
void USB_Force(BYTE bState)	Forces a J, K, or SE0 State on the USB D+/D- pins. Normally used for remote wakeup. bState Parameters are: USB_FORCE_J0x02 USB_FORCE_K0x01 USB_FORCE_SE00x00 USB_FORCE_NONE0xFF Note: When using this API Function and GPIO pins from Port 1 (P1.2-P1.7), the application should use the Port_1_Data_SHADE shadow register to ensure consistent data handling. From assembly language, you can access the Port_1_Data_SHADE RAM location directly. From C language, you should include an extern reference: `extern BYTE Port_1_Data_SHADE;`
void USB_Suspend(void)	Puts the USB Transceiver into power-down mode, while maintaining the USB address assigned by the USB host. To restore the USB Transceiver to normal operation, the USB_Resume function should be called.
void USB_Resume(void)	Puts the USB Transceiver into normal operation, following a call to USB_Suspend. It retains the USB address that had been assigned by the USB host.

Table 2. Human Interface Device (HID) Class Support API

Function	Description
BYTE USB_UpdateHIDTimer(BYTE)	Updates the HID Report timer for the specified interface and returns 1 if the timer expired and 0 if not. If the timer expired, it reloads the timer.
BYTE USB_bGetProtocol(BYTE)	Returns the protocol for the specified interface

USB_Start

Description:

Performs all required initialization for USB Device User Module.

C Prototype:

```
void  USB_Start(BYTE bDevice)
```

Assembly:

```
lcall  USB_Start
```

Parameters:

Register A contains the device number from the desired Device Descriptor set that was entered with the USB Setup Wizard.

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_Stop

Description:

Performs all the shutdown tasks required for the USB User Module.

C Prototype:

```
void  USB_Stop(void)
```

Assembly:

```
lcall  USB_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_bCheckActivity

Description:

Checks for USB Bus Activity.

C Prototype:

```
BYTE  USB_bCheckActivity(void)
```

Assembly:

```
lcall  USB_bCheckActivity
```

Parameters:

None

Return Value:

Returns 1 if the USB was active since the last check, otherwise returns 0.

Side Effects:

You can alter the A and X registers by this function.

USB_bGetConfiguration

Description:

Gets the current configuration of the USB device.

C Prototype:

```
BYTE  USB_bGetConfiguration(void)
```

Assembly:

```
lcall  USB_bGetConfiguration
```

Parameters:

None

Return Value:

Returns the currently assigned configuration. Returns 0 if the device is not configured.

Side Effects:

You can alter the A and X registers by this function.

USB_bGetEPState

Description:

Gets the Endpoint state for the specified endpoint. The endpoint can have one of three states, two of the states mean different things for IN and OUT endpoints. The following table outlines the possible states and their meaning for IN and OUT endpoints.

C Prototype:

```
BYTE  USB_bGetEPState(BYTE bEP)
```

Assembly:

```
MOv  A, 1               ; Select endpoint 1
lcall  USB_bGetEPState
```

Parameters:

Register A contains the endpoint number

Return Value:

Returns the current state of the specified USB endpoint.

State	Value	Description
NO_EVENT_PENDING	0x00	Indicates that the endpoint is awaiting SIE action
EVENT_PENDING	0x01	Indicates that the endpoint is awaiting CPU action
NO_EVENT_ALLOWED	0x02	Indicates that the endpoint is locked from access
IN_BUFFER_FULL	0x00	The IN endpoint is loaded and the mode is set to ACK IN
IN_BUFFER_EMPTY	0x01	An IN transaction has occurred and more data can be loaded
OUT_BUFFER_EMPTY	0x00	The OUT endpoint is set to ACK OUT and is waiting for data
OUT_BUFFER_FULL	0x01	An OUT transaction has occurred and data can be read

Side Effects:

You can alter the A and X registers by this function.

USB_bGetEPAckState

Description:

Determines whether or not an ACK transaction occurred on this endpoint by reading the ACK bit in the control register of the endpoint. This function does not clear the ACK bit.

C Prototype:

```
BYTE USB_bGetEPAckState(BYTE bEPNumber)
```

Assembly:

```
MOV A, 1 ; Select endpoint 1
lcall USB_bGetEPAckState
```

Parameters:

Register A contains the endpoint number.

Return Value:

If an ACKed transaction occurred then this function returns a non-zero value. Otherwise a zero is returned.

Side Effects:

You can alter the A and X registers by this function.

USB_bRWUEnabled

Description:

This function determines whether the host has enabled REMOTE WAKEUP.

C Prototype:

```
BYTE  USB_bRWUEnabled(void)
```

Assembly:

```
lcall  USB_bRWUEnabled
```

Parameters:

None

Return Value:

Returns 1 if the USB host has enabled REMOTE WAKEUP, otherwise returns 0.

Side Effects:

You can alter the A and X registers by this function.

USB_bGetEPCount

Description:

This function returns the value of the endpoint count register. The Serial Interface Enginer (SIE) includes two bytes of checksum data in the count. This function subtracts two from the count before returning the value. This function should only be called for OUT endpoints after a call to USB_GetEPState returns EVENT_PENDING.

C Prototype:

```
BYTE  USB_bGetEPCount(BYTE)
```

Assembly:

```
MOV  A, 1             ; Select endpoint 1
lcall  USB_bGetEPCount
```

Parameters:

Register A contains the endpoint number

Return Value:

Returns the current byte count from the specified USB endpoint.

Side Effects:

You can alter the A and X registers by this function.

USB_XLoadEP

Description:

Loads and enables the specified USB endpoint for an IN transfer.

C Prototype:

```
void  USB_XLoadEP(BYTE*)
```

Assembly:

```
lcall  USB_XLoadEP
```

Parameters:

Register A contains the address of the RAM buffer containing the IN data

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_bReadOutEP

Description:

Moves the specified number of bytes from endpoint RAM to data RAM. The number of bytes actually transferred from endpoint RAM to data RAM is the lesser of the actual number of bytes sent by the host and the number of bytes requested by the wCount argument.

C Prototype:

```
BYTE USB_bReadOutEP(BYTE bEPNumber, BYTE * pData, BYTE bLength)
```

Assembly:

```
mov A, 8
push A
mov A, >pData
push A
mov A, <pData
push A
mov A, 1
push A
lcall USB_bReadOutEP
```

Parameters:

bEPNumber is the Endpoint Number

pData is a pointer to a data array to which the Data from the Endpoint space is loaded.

bLength is the number of bytes to transfer from the array and then sent as a result of an IN request.

Valid values are between 0 and 8. The function moves less than that if the number of bytes sent by the host are less than those requested.

Return Value:

Returns the number of bytes sent by the host to the USB device. This can be more or less than the number of bytes requested.

Side Effects:

You can alter the A and X registers by this function.

USB_EnableEP

Description:

Enables the specified USB OUT endpoint. This function should not be called for IN endpoints.

C Prototype:

```
void  USB_EnableEP(BYTE)
```

Assembly:

```
MOV  A, 1
lcall  USB_EnableEP
```

Parameters:

Register A contains the endpoint number

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_DisableEP

Description:

Disables the specified USB OUT endpoint. This function should not be called for IN endpoints.

C Prototype:

```
void  USB_DisableEP(BYTE)
```

Assembly:

```
MOV  A, 1            ; Select endpoint 1
lcall  USB_DisableEP
```

Parameters:

Register A contains the endpoint number

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_Force

Description:

Forces a J, K, or SE0 state on the D+/D-. Used for signaling remote wakeup to the USB host.

C Prototype:

```
void  USB_Force(BYTE)
```

Assembly:

```
MOV  A, USB_FORCE_K
lcall  USB_Force
```

Parameters:

Register A contains one of the following constants:.

State	Value	Description
USB_FORCE_SE0	0x00	Force a Single Ended 0 onto the D+/D- lines
USB_FORCE_J	0x02	Force a J State onto the D+/D- lines
USB_FORCE_K	0x01	Force a K State onto the D+/D- lines
USB_FORCE_NONE	0xFF	Return bus to SIE control

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_Suspend

Description:

Puts the USB transceiver in power down mode, while maintaining the USB address assigned by the USB host. To restore the USB transceiver to normal operation, call the USB_Resume function.

C Prototype:

```
void  USB_Suspend(void)
```

Assembly:

```
lcall  USB_Suspend
```

Parameters:

None

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_SetPowerStatus

Description:

Sets the current power status. Set the power status to one for self powered or zero for bus powered. The device replies to USB GET_STATUS requests based on this value. This allows the device to properly report its status for USB Chapter 9 compliance. Devices may change their power source from self powered to bus powered at any time and report their current power source as part of the device status. Call this function when your device changes from self powered to bus powered or vice versa, and set the status appropriately.

C Prototype:

```
void USB_SetPowerStatus(BYTE bPowerStaus);
```

Assembly:

```
MOV A, USB_DEVICE_STATUS_SELF_POWERED ; Select self powered
```

```
lcall USB_SetPowerStatus
```

Parameters:

bPowerStatus contains the desired power status, one for self powered or zero for bus powered. Symbolic names are given in C and assembly and their associated values are given in the following table:

State	Value	Description
USB_DEVICE_STATUS_BUS_POWERED	0x00	Set the device to bus powered.
USB_DEVICE_STATUS_SELF_POWERED	0x01	Set the device to self powered.

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_Resume

Description:

Puts the USB transceiver in normal operation, following a call to USB_Suspend. It retains the USB address that had been assigned by the USB host.

C Prototype:

```
void  USB_Resume(void)
```

Assembly:

```
lcall  USB_Resume
```

Parameters:

None

Return Value:

None

Side Effects:

You can alter the A and X registers by this function.

USB_UpdateHIDTimer

Description:

Updates the HID Report Idle timer and returns the expiry status. Reloads the timer if it expires.

C Prototype:

```
BYTE  USB_UpdateHIDTimer(BYTE bInterface)
```

Assembly:

```
MOV  A, 1                ; Select Interface 1
lcall  USB_UpdateHIDTimer
```

Parameters:

Register A contains the interface number

Return Value:

Register A contains:.

State	Value	Description
USB_IDLE_TIMER_EXPIRED	0x01	The timer has expired
USB_IDLE_TIMER_RUNNING	0x02	The timer is running
USB_IDLE_TIMER_INDEFINITE	0x00	The report should be sent upon a data/state change

Side Effects:

You can alter the A and X registers by this function.

USB_bGetProtocol

Description:

Returns the protocol value for the selected interface

C Prototype:

```
BYTE  USB_bGetProtocol(BYTE bInterface)
```

Assembly:

```
MOV  A, 1                ; Select Interface 1
lcall  USB_bGetProtocol
```

Parameters:

bInterface contains the desired interface number

Return Value:

Register A contains the protocol value

Side Effects:

You can alter the A and X registers by this function.

Sample Firmware Source Code

Sample code is available in the PSoC Designer Examples directory.

The following C code shows you how to use the USB User Module in a simple HID application. When it is connected to a PC host, the device is enumerated as a 3-button mouse. When the code is run, the mouse cursor zigzags from right to left.

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user mModules

BYTE abMouseData[3] = {0,0,0};
BYTE i = 0;

void main (void)
```

```
{
    M8C_EnableGInt; //Enable Global Interrupts

USB_Start(0); //Start USB Operation using device 0

while (!USB_bGetConfiguration()); //Wait for Device to enumerate
    USB_LoadEP (1, abMouseData, 3); //Begin initial USB transfers
    while(1)
    {
        if (USB_bGetEPAckState (1)) //Check and see if ACK has occured
        {
            USB_LoadEP (1, abMouseData, 3); //Load EP1 with mouse data

        if(i==128)
                abMouseData[1] = 0x05; //Start moving the mouse to the right
            else if(i==255)
                abMouseData[1] = 0xFB; //Start moving the mouse to the left
            i++;
        }
    }
}
```

The same project in Assembly is:

```
include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all user modules

export _main

area bss(RAM)      ; inform assembler that variables follow

abMouseData: blk 3 ; USB data variable
i:           blk 1 ; count variable

area text(ROM,REL) ; inform assembler that program code follows

_main:

    M8C_EnableGInt   ; Enable Global Interrupts

mov    A, 0
    lcall  USB_Start ; Start USB Operation using device 0

; Wait for Device to enumerate
.no_device:
    lcall  USB_bGetConfiguration
    cmp    A, 0
    jz     .no_device
    ; Enumeration is completed, load endpoint 1

mov    [USB_APIEPNumber], 1
    mov    [USB_APICount], 3
    mov    X, <abMouseData
    lcall  USB_XLoadEP
```

```
loop:
    mov     A, 1
    lcall   USB_bGetEPState
    cmp     A, IN_BUFFER_EMPTY
    jnz      loop
    ; ACK has occurred, load the endpoint

    mov     [USB_APIEPNumber], 1
    mov     [USB_APICount], 3
    mov     X, <abMouseData
    lcall   USB_XLoadEP

cmp     [i], 128
    jnz     .MoveLeft

    ; Start moving the mouse to the right
mov     [abMouseData+1], 5
    jmp     .Inc_i

.MoveLeft:
    ; Start moving the mouse to the left
cmp     [i], 255
    jnz     .Inc_i
    mov     [abMouseData+1], 251

.Inc_i:
    inc     [i]
    jmp     loop
```

Appendices

The following sections provide information on the USB specification and the USB setup wizard.

USB Standard Device Requests

The following section describes the requests supported by the USB User Module. If a request is not supported, the USB User Module normally responds with a STALL, indicating a Request Error.

Standard Device Request	USB User Module Support Description	USB 2.0 Spec Section
CLEAR_FEATURE	Device:	9.4.1
	Interface: Not supported.	
	Endpoint	
GET_CONFIGURATION	Returns the current device configuration value.	9.4.2
GET_DESCRIPTOR	Returns the specified descriptor.	9.4.3
GET_INTERFACE	Returns the selected alternate interface setting for the specified interface.	9.4.4
GET_STATUS	Device: Supported	9.4.5
	Interface: Supported	
	Endpoint: Supported	
SET_ADDRESS	Sets the device address for all future device accesses.	9.4.6
SET_CONFIGURATION	Sets the device configuration.	9.4.7
SET_DESCRIPTOR	This optional request is not supported.	9.4.8
SET_FEATURE	Device: DEVICE_REMOTE_WAKEUP support is selected by the bRemoteWakeUp User Module Parameter. TEST_MODE is not supported.	9.4.9
	Interface: Not supported.	
	Endpoint: The specified Endpoint is halted.	
SET_INTERFACE	Not supported.	9.4.10
SYNCH_FRAME	Because enCoRe II does not contain ISOC endpoints, this request is not supported.	9.4.11

HID Class Request

Class Request	USB User Module Support Description	Device Class Definition for HID - Section
GET_REPORT	Allows the host to receive a report by way of the Control pipe.	7.2.1
GET_IDLE	Reads the current idle rate for a particular Input report.	7.2.3
GET_PROTOCOL	Reads which protocol is currently active (either the boot or the report protocol).	7.2.5
SET_REPORT	Allows the host to send a report to the device, possibly setting the state of input, output, or feature controls.	7.2.2
SET_IDLE	Silences a particular report on the Interrupt In pipe until a new event occurs or the specified amount of time passes.	7.2.4
SET_PROTOCOL	Switches between the boot protocol and the report protocol (or vice versa).	7.2.6

USB Setup Wizard

This section describes all the USB descriptors provided by the USB User Module. The descriptions include the descriptor format and how user module parameters map into the descriptor data.

The Setup Wizard is a tool provided by Cypress to assist engineers in the designing of USB devices. The Setup Wizard displays the device descriptor tree; when expanded, the following folders that are part of the standard USB descriptor definitions appear:

- Device Attributes
- Configuration Descriptor
- Interface Descriptor
- HID Class Descriptor
- Endpoint Descriptor
- String/LANGID
- HID Descriptor

To access the setup wizard, right click the USB User Module icon in the device editor and click the USB Setup Wizard... menu item.

When the Device Descriptor tree is fully expanded, you see all the wizard options. The left side displays the name of the descriptor, the center displays the data, and the right displays the operation that may be performed for a particular descriptor. In some instances, a descriptor has a pull-down menu that presents available options.

Descriptor	Data		Operations
USB User Module descriptor root	"Device Name"		Add Device
Device descriptor	DEVICE_1		Remove|Add Configuration
Device attributes			
Vendor ID	FFFF		
Product ID	FFFF		
Device release (bcdDevice)	0000		
Device class	Undefined	pull-down	
Subclass	No Subclass	pull-down	
Protocol	None	pull-down	
Manufacturer string	No String	pull-down	
Product string	No String	pull-down	
Serial number string	No String	pull-down	
Configuration descriptor	CONFIG_NAME		Remove|Add Interface
Configuration attributes			
Configuration string	No String	pull-down	
Max power	100		
Device power	Bus Powered	pull-down	
Remote wakeup	Disabled	pull-down	
Interface descriptor	INTERFACE_NAME		Remove|Add Endpoint
Interface attributes			
Interface string	No String	pull-down	
Class	Vendor Specific	pull-down	
Subclass	No Subclass	pull-down	
HID class descriptor			
Descriptor type	Report	pull-down	
Country code	Not Supported	pull-down	
HID report	None	pull-down	
Endpoint descriptor	ENDPOINT_NAME		Remove
Endpoint attributes			
Endpoint number	0		
Direction	IN	pull-down	

Descriptor	Data		Operations
Transfer type	CNTRL	pull-down	
Interval	10		
Max packet size	8		
String/LANGID			
String descriptors	"Device Name"		Add String
LANGID		pull-down	
String	"Selected String Name"		Remove
Descriptor			
HID report root	"Device Name"		Add HID Report

Understanding the Setup Wizard

The Setup Wizard window may be described as a table that presents three major areas for programming. The first area is the Descriptor USB User Module, the second is the String/LANGID, and the third is the Descriptor HID Report. Use the two buttons below the table to perform the selected command.

The first section presents the Descriptor USB User Module. The second section presents the String/LANGID; when a string ID is required, this area is used to input that string. To add a string for a USB device, click on the **Add String** operation. The software adds a row and prompts you to edit your string here. Type the new string then click **Save/Generate**. Once the string is saved, it is available for use in the Descriptor USB User Module from the pull-down menus. If you close without saving, the string is lost.

The third area presents the Descriptor HID Report Root. From here you can request an HID Report for the selected device.

Descriptor USB User Module

The first column displays folders that may be expanded and collapsed. For the purpose of this discussion, the tree must be fully expanded so that all options are visible. The Setup Wizard permits entering data into the middle Data column; if there is a pull-down menu, it can be used to select a different option. If there is no pull-down menu, but there is "data", use the cursor to highlight and select the "data", then overwrite that data with another value or text option. All the values entered must meet the USB 2.0 Chapter 9 Specifications.

The first folder that is displayed at the top is the USB User Module Descriptor Root. It has the user module name in the Data column (this is the user module name given to it by the software; “Rename" is one of the options when you right-click on the user module). This user module was placed in the Interconnect View. The Add Device operation on the right-hand column adds another USB device complete with all the different fields required for describing it. The new USB device will be listed at the bottom after the Endpoint Descriptor. Click the Save/Generate Descriptors button. If you do not save the newly added device, it will not be available for use.

Device Descriptor has DEVICE_NUMBER as the Data; it may be removed or a configuration may be added. All the information about a particular USB device may be entered by overwriting the existing data or by using a pull-down menu.

When the input of data is complete, either by using the pull-down menus or by filling in alphanumeric text in the appropriate spots, select the Save/Generate Descriptors button; the data that defines your design can only be made permanent by the use of this button.

USB Suspend, Resume, and Remote Wakeup

The USB User Module supports USB Suspend, Resume, and Remote Wakeup. Since these features must be tightly coupled into the user application, the USB User Module provides a set of API Functions.

USB Activity Monitoring

The USB_bCheckActivity API function provides a means to check if any USB Bus activity has occurred. The application uses the function to determine if the conditions to enter USB Suspend have been met.

USB Suspend

After the conditions to enter USB Suspend are met, the application should take appropriate steps to reduce current consumption to meet the suspend current requirements. To put the USB SIE and transceiver into power down mode, the application calls the USB_Suspend API function. This function disables the USB block but maintains the current USB address (in the USBCR register). The device can use the Sleep feature of the enCoRe II to reduce current consumption.

USB Resume

While the device is suspended, it should periodically check to determine if the conditions to leave the suspended state have been met. One way to check resume conditions is to use the Sleep Timer to periodically wake the device. If the resume conditions are met, the application calls the USB_Resume API function. This function enables the USB SIE and transceiver, bringing them out of power down mode. It does not change the USB address field of the USBCR register, thus maintaining the USB address previously assigned by the host.

USB Remote Wakeup

If the device supports remote wakeup, the application can determine if the host has enabled remote wakeup with the USB_bRWUEnabled API function. When the device is suspended and it determines the conditions to initiate a remote wakeup are met, the application can use the USB_Force API function to force the appropriate J and K states onto the USB Bus, thus signaling a remote wakeup.

Creating Vendor Specific Device Requests and Overriding Existing Requests

The USB User Module supports vendor specific requests (VSR) and overriding existing requests by providing a SETUP request dispatcher, exposing the control endpoint initialization entry points for Control Read, Control Write, and No Data control transfers and providing transfer completion notification. The USB User Module Endpoint 0 Interrupt Service Routine (ISR) transfers control to a handler provided by the application. The handler initializes specific user module data structures and transfers control back to the Endpoint 0 ISR. The user module handles the subsequent data and status stages of the transfer without any involvement of the user application. Upon completion of the transfer, the user module updates a completion status block. The status block can be monitored by the application to determine it the VSR has completed.

Vendor Specific Request Processing

All control transfers, including VSRs, have a SETUP stage, an optional data stage, and a status stage. The SETUP stage is unique for each request. During the SETUP stage, the handler must prepare the application to receive data for control writes or prepare data for transmission to the host for control reads. For no data control transfers, the handler may extract information from the SETUP packet itself.

The Endpoint 0 ISR processes the data and status stages exactly the same way for all requests. During the data stage, the ISR copies data to or from the control endpoint buffer (registers EP0DATA0-EP0DATA7) depending upon the direction of the transaction.

During the status stage, the Endpoint 0 ISR updates the completion status block. The ISR detects and reports premature completion.

Vendor Specific Request Dispatch Routines

Depending on the application requirements, the Endpoint 0 ISR can dispatch up to eight types of VSRs based on the bmRequestType field of the SETUP. Refer to section 9.3 of the USB 2.0 specification for a discussion of bmRequestType.

Table 3. Vendor Specific Request Dispatch Routine Names

Direction	Recipient	Dispatch Routine Entry Point	Enable Flag
Host to Device (Control Write)	Device	USB_DT_h2d_vnd_dev_Dispatch	USB_CB_h2d_vnd_dev
	Interface	USB_DT_h2d_vnd_ifc_Dispatch	USB_CB_h2d_vnd_ifc
	Endpoint	USB_DT_h2d_vnd_ep_Dispatch	USB_CB_h2d_vnd_ep
	Other	USB_DT_h2d_vnd_oth_Dispatch	USB_CB_h2d_vnd_oth
Device to Host (Control Read)	Device	USB_DT_d2h_vnd_dev_Dispatch	USB_CB_d2h_vnd_dev
	Interface	USB_DT_d2h_vnd_ifc_Dispatch	USB_CB_d2h_vnd_ifc
	Endpoint	USB_DT_d2h_vnd_ep_Dispatch	USB_CB_d2h_vnd_ep
	Other	USB_DT_d2h_vnd_oth_Dispatch	USB_CB_d2h_vnd_oth

Follow these steps for an application to provide an assembly language dispatch routine for the vendor specific device request:

1. In the *USB.inc* file, enable support for the vendor specific dispatch routine. Find the dispatch routine enable flag and set EQU to 1.
2. Write an appropriately named assembly language routine to handle the device request. Use the entry points listed in the table above.

Override Existing Request Routines

To override a standard or class specific device request, or enable an unsupported device request, follow these steps:

1. In the *USB.inc* file, redefine the specific device request as USB_APP_SUPPLIED.
2. Write an appropriately named assembly language function to handle the device request. The name of the assembly language function is APP_ plus the device name.

For example, to override the supplied HID class Set Report request, USB_CB_SRC_h2d_cls_ifc_09, enable the routine with these changes to *USB.inc*:

```
;@PSoC_UserCode_BODY_1@ (Do not change this line.)
   ;---------------------------------------------------
   ; Insert your custom code below this banner
   ;---------------------------------------------------
   ;   NOTE: interrupt service routines must preserve
   ;   the values of the A and X CPU registers.

; Enable an override of the HID class Set Report request.
USB_CB_SRC_h2d_cls_ifc_09: EQU USB_APP_SUPPLIED

   ;---------------------------------------------------
   ; Insert your custom code above this banner
   ;---------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
```

Then, write an assembly language routine named APP_USB_CB_SRC_h2d_cls_ifc_09. Device request names are derived from the USB bmRequestType and bRequest values (USB spec**ification** Table 9-2).

This code is a stub for the assembly routine for the previous example:

```
export APP_USB_CB_SRC_h2d_cls_ifc_09
APP_USB_CB_SRC_h2d_cls_ifc_09:

;Add your code here.

; Long jump to the appropriate return entry point for your application.
LJMP USBFS_InitControlWrite
```

Customizing the HID Class Report Storage Area

If you enable optional HID class support, the Setup Wizard creates a fixed-size report storage area for data reports from the HID class device. It creates separate report areas for IN, OUT, and FEATURE reports. This area is sufficient for the case where no Report ID item tags are present in the Report descriptor and therefore only one Input, Output, and Feature report structure exists. If you want better control over the report storage size or want to support multiple report IDs, you must do the following:

1. Use the wizard to specify your device description, endpoints, and HID reports then generate the application.
2. Disable the wizard defined report storage area in *USBUM_descr.asm*.
3. Copy the wizard created code that defines the report storage area.
4. Paste it into the protected user code area in *USBUM_descr.asm* or a separate assembly language file.
5. Customize the code to define the report storage area.

Dispatch and Override Routine Requirements.

At a minimum, the dispatch or override routine must return control back to the Endpoint ISR by a LJMP to one of the Endpoint 0 ISR Return Points listed in the following table. The dispatch routine may destroy the A and X registers, but the Stack Pointer (SP) and any other relevant context must be restored prior to returning control to the ISR.

Table 4. Endpoint 0 ISR Return Points

Return Entry Point	Required Data Items	Description
USB_Not_Supported	This return point should be used when the request is not supported. It STALLs the request.	
	Data Items: None	
USB_InitControlRead	This return point is used to initiate a Control Read transfer.	
	USB_DataSource	The data source is RAM or ROM (USB_DS_RAM or USB_DS_ROM). This is necessary since different instructions must be used to move the data from the source ROMX or MOV.
	USB_TransferSize (WORD)	The number of data bytes to be transferred
	USB_DataPtr (WORD)	RAM or ROM address of the data
	USB_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.
USB_InitControlWrite	This return point is used to initiate a Control Write transfer.	
	USB_DataSource	USB_DS_RAM (the destination for control writes must RAM).
	USB_TransferSize (WORD)	Size of the application buffer to receive the data
	USB_DataPtr (WORD)	RAM address of the application buffer to receive the data
	USB_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.
USB_InitNoDataControlTransfer	This return point is used to initiate a No Data Control transfer.	
	USB_StatusBlockPtr (WORD) optional	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.

The status completion block contains two data items, a one byte completion status code and a two byte transfer length. The "main" application can monitor the completion status to determine how to proceed. Completion status codes are listed in the following table. The transfer length is the actual number of data bytes transferred.

Table 5. USB Transfer Completion Codes

Completion Code	Description
USB_XFER_IDLE (0x00)	USB_XFER_IDLE indicates that the associated data buffer does not have valid data and the application should not use the buffer. The actual data transfer will take place while the completion code is USB_XFER_IDLE, although it does not indicate a transfer is in progress.
USB_XFER_STATUS_ACK (0x01)	USB_XFER_STATUS_ACK indicates the control transfer status stage completed successfully. At this time, the application can use the associated data buffer and its contents.
USB_XFER_PREMATURE (0x02)	USB_XFER_PREMATURE indicates that the control transfer was interrupted by the SETUP of a subsequent control transfer. For control writes, the contents of the associated data buffer will contain the data up to the premature completion.
USB_XFER_ERROR (0x03)	USB_XFER_ERROR indicates that the expected status stage token was not received.

Specify Your Device and Generate Application

Use the USB setup wizard to specify your device description, endpoints, and HID reports. Click the **Generate Application** button in PSoC Designer.

Disable the Wizard Defined Report Storage Area

In the *USBUM_descr.asm* file, disable the wizard defined storage area by uncommenting the WIZARD_DEFINED_REPORT_STORAGE line in the custom code area as shown.

```
WIZARD: equ 1
WIZARD_DEFINED_REPORT_STORAGE: equ 1
   ;-------------------------------------------------
;@PSoC_UserCode_BODY_1@ (Do not change this line.)
   ;-------------------------------------------------
   ; Insert your custom code below this banner
   ;-------------------------------------------------
   ; Redefine the WIZARD equate to 0 below by
; uncommenting the WIZARD: equ 0 line
; to allow your custom descriptor to take effect
   ;-------------------------------------------------

; WIZARD: equ 0
   WIZARD_DEFINED_REPORT_STORAGE: equ 0

   ;-------------------------------------------------
   ; Insert your custom code above this banner
   ;-------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
```

Copy the Wizard Created Code

Find this code in *USBUM_descr.asm*.

```
;-------------------------------------------------------------------------
; HID IN Report Transfer Descriptor Table for ()
;-------------------------------------------------------------------------
IF WIZARD_DEFINED_REPORT_STORAGE
AREA  func_lit     (ROM,REL,CON)
.LITERAL
USBUM_D0_C1_I0_IN_RPTS:
TD_START_TABLE 1                                  ; Only 1 Transfer Descriptor
  TD_ENTRY     USB_DS_RAM, USBUM_HID_RPT_3_IN_RPT_SIZE, USBUM_INTERFACE_0_IN_RPT_DATA,
NULL_PTR
.ENDLITERAL
ENDIF ; WIZARD_DEFINED_REPORT_STORAGE
```

There are three sections, one each for the IN, OUT, and FEATURE reports. Copy all three sections.

Paste the Code Into the Protected User Code Area

You can paste the code into the protected user code area of *USBUM_descr.asm* shown or a separate assembly language file.

```
   ;-----------------------------------------------
   ;@PSoC_UserCode_BODY_2@ (Do not change this line.)
   ;-----------------------------------------------
   ; Redefine your descriptor table below. You might
   ; cut and paste code from the WIZARD descriptor
   ; above and then make your changes.
   ;-----------------------------------------------

   ;-----------------------------------------------
   ; Insert your custom code above this banner
   ;-----------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)
; End of File USBUM_descr.asm
```

Customize the Code to Define the Report Storage Area

To define the report storage area, you write your own descriptor table entries. The table contains entries to define storage space for the required data items. Each transfer descriptor entry in the table creates a new Report ID. IDs are numbered consecutively, starting with zero. Report ID 0 is not used; you cannot specify a Report ID of 0, but the transfer descriptor entry specified for the ID 0 is used in the case that no Report IDs are present in the Report descriptor. For the sake of code efficiency, you should use Report IDs in order starting with ID 1.

Table 6. Transfer Descriptor Table Entries

Table Entry	Required Data Items	Description
TD_START_TABLE	USB_NumberOfTableEntries	Number of Report IDs defined. IDs are numbered consecutively from 0. Report ID 0 is not used.
TD_ENTRY		
	USB_DataSource	The data source is RAM or ROM (USB_DS_RAM or USB_DS_ROM).
	USB_TransferSize	Size of the data transfer in bytes. The first byte is the Report ID.
	USB_DataPtr	RAM or ROM address of the data transfer.
	USB_StatusBlockPtr	Address of a status block allocated with the USB_XFER_STATUS_BLOCK macro.

The following example sets up the unused Report ID 0, and two other IN reports with different sizes. Note Conditional assembly statements are only necessary if you place the code in the protected user code area of *USBUM_descr.asm.*

```
;-------------------------------------------------------------------------
; HID IN Report Transfer Descriptor Table for ()
;-------------------------------------------------------------------------
IF WIZARD_DEFINED_REPORT_STORAGE
ELSE

_ID0_RPT_SIZE:   EQU 8       ; 7 data bytes + report ID = 8 bytes (unused)
_SM_RPT_SIZE:    EQU 3       ; 2 data bytes + report ID = 3 bytes
_LG_RPT_SIZE:    EQU 5       ; 4 data bytes + report ID = 5 bytes

AREA data (RAM, REL, CON)

EXPORT _ID0_RPT_PTR
_ID0_RPT_PTR: BLK 8         ; Allocates space for report ID0 (unused)
EXPORT _SM_RPT_PTR
_SM_RPT_PTR:    BLK 3       ; Allocates space for report ID1
EXPORT _LG_RPT_PTR
_LG_RPT_PTR:    BLK 5       ; Allocates space for report ID2

AREA bss (RAM, REL, CON)

EXPORT _SM_RPT_STS_PTR
_SM_RPT_STS_PTR: USB_XFER_STATUS_BLOCK
EXPORT _LG_RPT_STS_PTR
_LG_RPT_STS_PTR: USB_XFER_STATUS_BLOCK

AREA  func_lit     (ROM,REL,CON)
.LITERAL
EXPORT USBUM_D0_C1_I0_IN_RPTS:
  TD_START_TABLE 3
```

```
  TD_ENTRY  USB_DS_RAM, _ID0_RPT_SIZE, _ID0_RPT_PTR, NULL_PTR ; ID0 unused
  TD_ENTRY  USB_DS_RAM, _SM_RPT_SIZE, _SM_RPT_PTR, _SM_RPT_STS_PTR ; ID1
  TD_ENTRY  USB_DS_RAM, _LG_RPT_SIZE, _LG_RPT_PTR, _LG_RPT_STS_PTR ; ID2
.ENDLITERAL

ENDIF ; WIZARD_DEFINED_REPORT_STORAGE
```

Version History

Version	Originator	Description
1.6	DHA	Added Version History.
1.70	DHA	Added list of HID templates to wizard.
1.80	DHA	1 Added wizard help file. 2. Added verification of the writing EP0_CR. 3. Added verification of the SIE MODEs and ACK bit into the EP0 ISR. 4. Updated the constant area definition in user module. 5. Removed directives .SECTION and .ENDSECTION for USB_bGetProtocol and USB_UpdateHIDTimer functions.
1.90	DHA	1. Added initialization of the USB_Protocol variable to comply with HID specifications. 2. Added USB_bGetEPAckState(), USB_bReadOutEP(), and USB_SetPowerStatus() API functions.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2004-2014 Cypress Semiconductor Corporation. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.