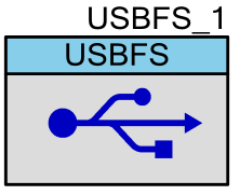
Full Speed USB (USBFS)

2.80



Features

- USB Full Speed device interface driver
- Support for interrupt, control, bulk, and isochronous transfer types
- Run-time support for descriptor set selection
- Optional USB string descriptors
- Optional USB HID class support
- Optional Bootloader support
- Optional Audio class support (See the USBFS Audio section)
- Optional MIDI devices support (See the USBFS MIDI section)
- Optional communications device class (CDC) support (See the USBUART section)

General Description

The USBFS component provides a USB full-speed Chapter 9 compliant device framework. It provides a low-level driver for the control endpoint that decodes and dispatches requests from the USB host. Additionally, this component provides a USBFS customizer to make it easy to construct your descriptor.

You have the option of constructing a HID-based device or a generic USB Device. Select HID (and switch between HID and generic) by setting the Configuration/Interface descriptors.

Refer to the USB-IF device class documentation for additional information on descriptors (http://www.usb.org/developers/devclass/).

Note Cypress offers a set of USB development tools, called SuiteUSB, available free of charge when used with Cypress silicon. You can obtain SuiteUSB from the Cypress website: http://www.cypress.com.

When to Use a USBFS

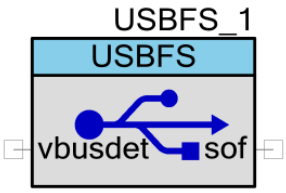
Use the USBFS component when you want to provide your application with a USB 2.0 compliant device interface.

Quick Start

1. Drag a USBFS component from the Component Catalog onto your design.
2. Notice the clock errors in the Notice List window; double-click on an error to open the System Clock Editor.
3. Configure the following clocks:
   a) **ILO**: Select 100 kHz.
   b) **IMO**: Select Osc 24.000 MHz.
   c) **USB**: Enable and select IMOx2 – 48.000 MHz.
4. Select **Build** to generate APIs.

Input/Output Connections

This section describes the input and output connections for the USBFS. An asterisk (*) in the list of I/Os indicates that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.



sof – Output *

The start-of-frame (sof) output allows endpoints to identify the start of the frame and synchronize internal endpoint clocks to the host. This output is visible if the **Enable SOF Output** parameter in the **Advanced** tab of the customizer is selected.

vbusdet – Input *

The vbusdet input provides the ability to connect VBUS for power monitoring. This input is visible if the **Enable VBUS Monitoring** and **External VBUS** parameters in the **Advanced** tab of the customizer are selected.

Component Parameters

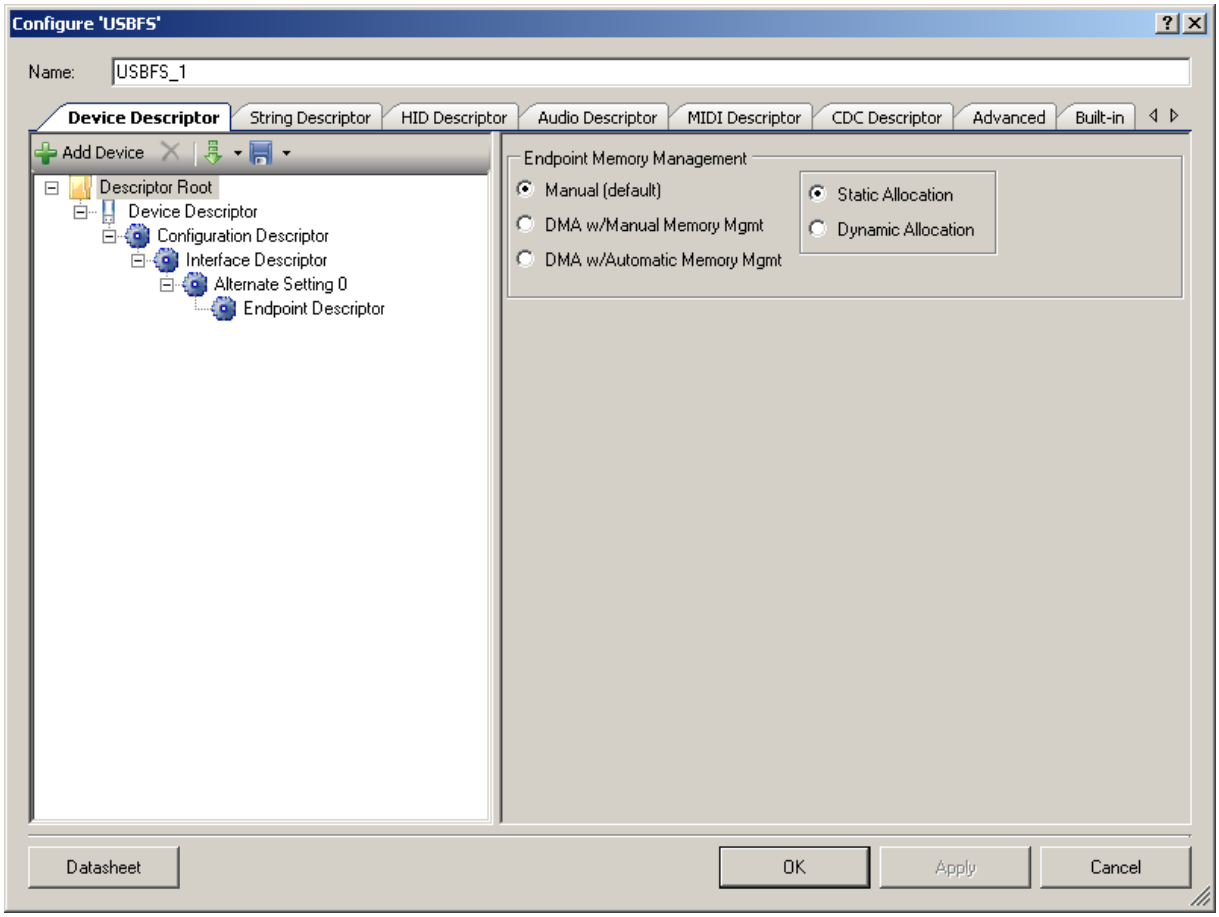
The USBFS component is driven by information generated by the USBFS Configure dialog. This dialog, or “customizer,” facilitates the construction of the USB descriptors and integrates the information generated into the driver firmware used for device enumeration.

The USBFS component does not function without first running the wizard and selecting the appropriate attributes to describe your device. The code generator takes your device information and generates all of the needed USB descriptors.

To begin, drag a USBFS component onto your design and double-click it to open the Configure USBFS dialog. The Configure USBFS dialog contains the following tabs and settings:

Device Descriptor Tab

Descriptor Root



Import and Save Tool Buttons

The **Save** button allows you to save information about the component configuration into an XML configuration file. In the drop-down list you can choose either **Save Current Descriptor** or **Save Root Descriptor**. The first option saves the configuration of the selected descriptor. The second option saves the whole device descriptor tree.

The **Import** button allows you to import the descriptor configuration. In the drop-down list you can choose either **Import Current Descriptor** or **Import Root Descriptor**. The first option loads the configuration of the selected descriptor. The second option loads the tree of descriptors. In this case, previously configured descriptors are not removed.

Note The same **Import** and **Save** tool buttons are present on the other descriptors tabs: **HID Descriptor**, **Audio Descriptor**, and **CDC Descriptor**. They are used to save and import descriptor configurations that are configured on those tabs.

Endpoint Memory Management

The USBFS block contains 512 bytes of target memory for the data endpoints to use. However, the architecture supports a cut-through mode of operation (DMA w/Automatic Memory Management) that reduces the memory requirement based on system performance.

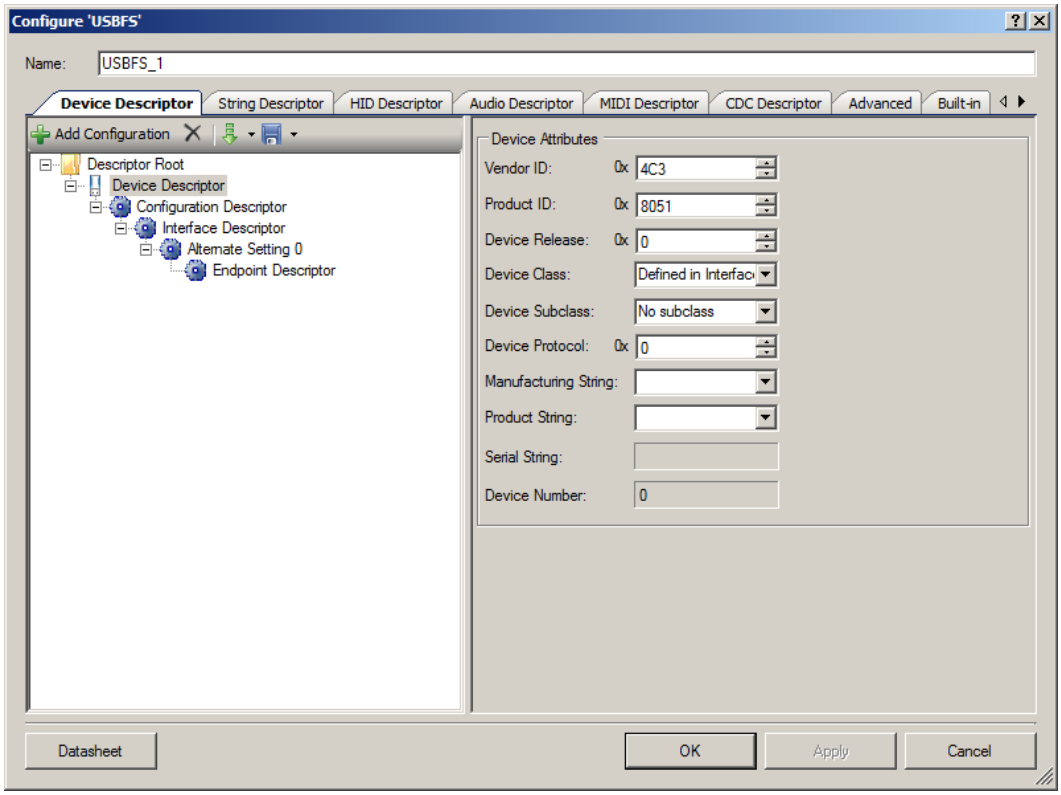
Some applications can benefit from using Direct Memory Access (DMA) to move data into and out of the endpoint memory buffers.

- **Manual** (default) – Select this option to use LoadInEP/ReadOutEP to load and unload the endpoint buffers.
  - **Static Allocation** – The memory for the endpoints is allocated immediately after a SET_CONFIGURATION request. This takes longest when multiple alternate settings use the same endpoint (EP) number.
  - **Dynamic Allocation** – The memory for the endpoints is allocated dynamically after each SET_CONFIGURATION and SET_INTERFACE request. This option is useful when multiple alternate settings are used with mutually exclusive EP settings.
- **DMA w/Manual Memory Management** – Select this option for manual DMA transactions. The LoadInEP/ReadOutEP functions fully support this mode and initialize the DMA automatically.
- **DMA w/Automatic Memory Management** – Select this option for automatic DMA transactions. This is the only configuration that supports combined data endpoint use of more than 512 bytes. Use the LoadInEP/ReadOutEP functions for initial DMA configuration.

PSoC does not support DMA transactions directly between USB endpoints and other peripherals. All DMA transactions involving USB endpoints (in and out) must terminate or originate with main system memory.

Applications requiring DMA transactions directly between USB endpoints and other peripherals must use two DMA transactions. The two transactions move data to main system memory as an intermediate step between the USB endpoint and the other peripheral.

Device Descriptor



Device Attributes

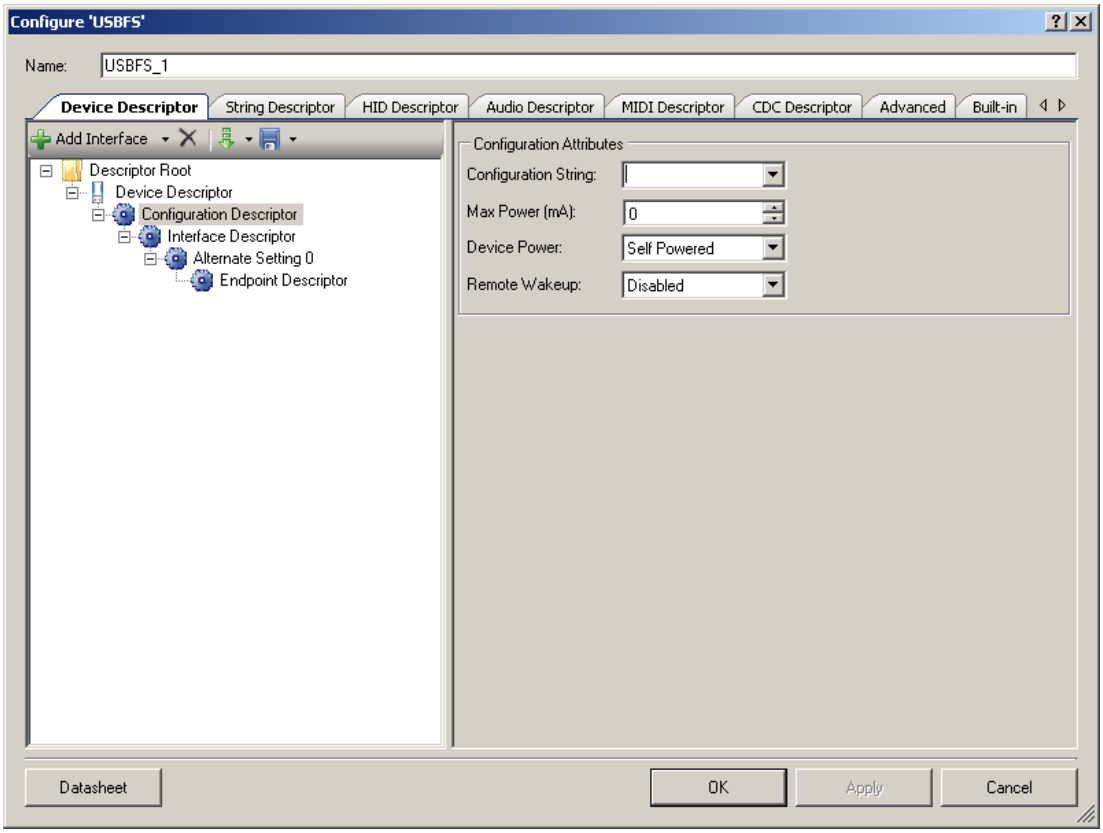
- **Vendor ID** – Your company USB vendor ID (obtained from USB-IF)

  Note Vendor ID 0x4B4 is a Cypress-only VID and may be used for development purposes only. Products cannot be released using this VID; you must obtain your own VID.

- **Product ID** – Your specific product ID
- **Device Release** – Your specific device release (device ID)
- **Device Class** – Device class is defined in **Interface Descriptor**, **CDC**, or **Vendor-Specific**
- **Device Subclass** – Dependent upon **Device Class**
- **Device Protocol**

- **Manufacturing String** – Manufacturer-specific description string to be displayed when the device is attached.
- **Product String** – Product-specific description string to be displayed when the device is attached.
- **Serial String**
- **Device Number** – Index number of the device in the array of devices.

Configuration Descriptor



Configuration Attributes

- **Configuration string**
- **Max Power (mA)** – Enter the maximum power consumption of the USB device from the bus when the device is fully operational, in this specific configuration.

  Note The **Device Power** parameter reports whether the configuration is bus powered or self powered. Device status reports whether the device is currently self powered. If a device is disconnected from its external power source, it updates device status to indicate

that it is no longer self powered. A device cannot increase its power draw from the bus, when it loses its external power source, beyond the amount reported by its configuration.

- **Device Power** – **Bus Powered** or **Self Powered** device. The USBFS does not support both settings simultaneously.
- **Remote Wakeup** – **Enabled** or **Disabled**

Interface Association Descriptor

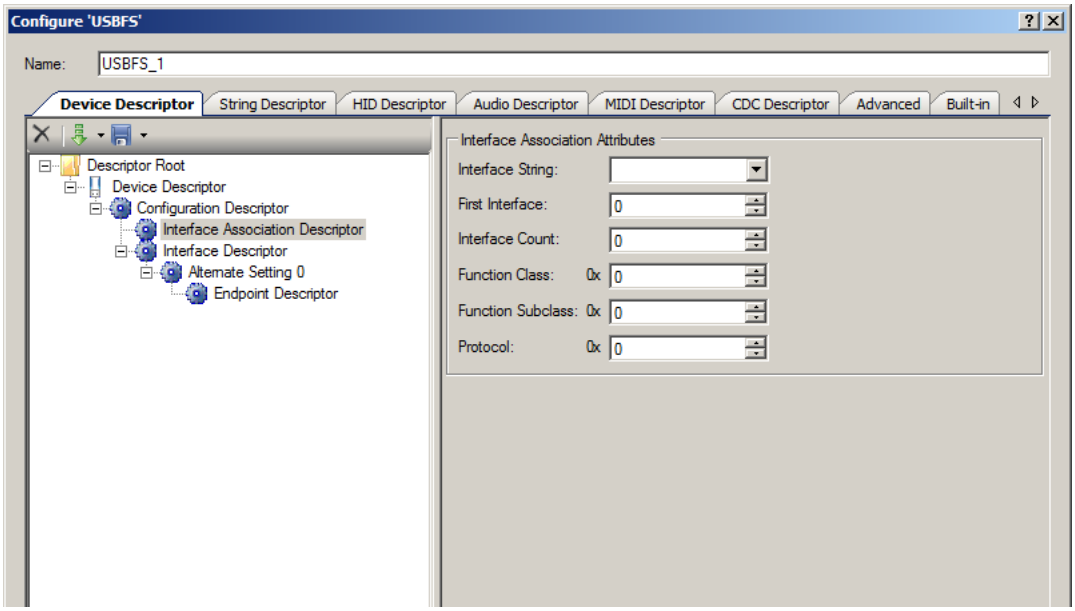
Interface Association Descriptor (IAD) is used to group multiple interfaces, in a multi-function device, to the one logical device function.

Devices that use the IAD must use the device class, subclass, and protocol codes as defined in the following table. This set of class codes is defined as the *Multi-Interface Function Device Class Codes*.

Device Attributes	Value	Description
Device Class	0xEF	Miscellaneous Device Class
Device Subclass	0x02	Common Class
Device Protocol	0x01	Interface Association Descriptor

To Add Interface Association Descriptor:

1. Select **Configuration Descriptor** item in the **Descriptor Root** tree.
2. Click **Add Interface** tool button, select **Association**

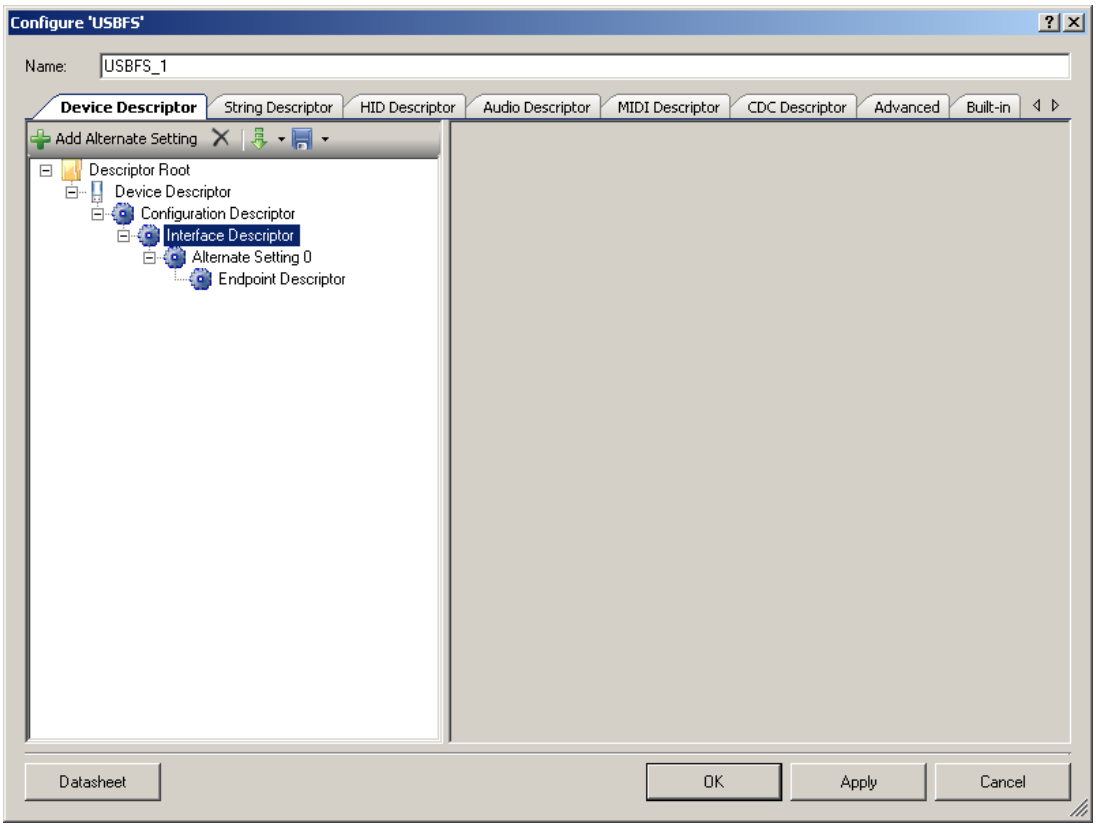


Interface Association Attributes

- **Interface String** – Index of string descriptor describing this function.
- **First Interface** – Interface number of the first interface associated with this function.
- **Interface Count** – Number of contiguous interfaces associated with this function.
- **Function Class** – Class code. Usually the same value as Class value in the first associated interface.
- **Function Subclass** – Subclass code.
- **Protocol** – Protocol code.

Interface Descriptor

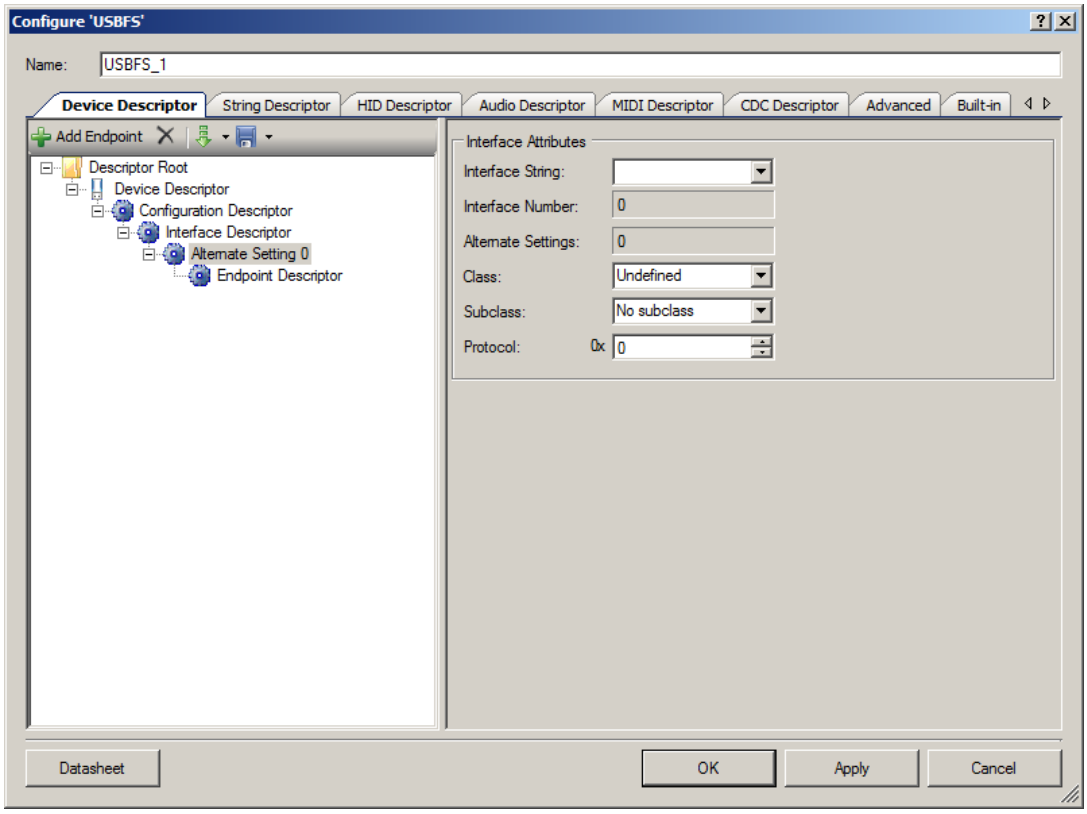
This level is used to add and delete Interface Alternate Settings. The interfaces are configured in the Alternate Setting.



Alternate Setting 0 is automatically provided to configure your device. If your device uses isochronous endpoints, note that the USB 2.0 specification requires that no device default interface settings can include any isochronous endpoints with nonzero data payload sizes. This is specified using **Max Packet Size** in the **Endpoint Descriptor**.

For isochronous devices, use an alternate interface setting other than the default Alternate Setting 0 to specify nonzero data payload sizes for isochronous endpoints. Additionally, if your isochronous endpoints have a large data payload, you should use additional alternate configurations or interface settings to specify a range of data payload sizes. This increases the chance that the device can be used successfully in combination with other USB devices.

Interface Descriptor—Alternate Settings



Interface Attributes

- **Interface String**
- **Interface Number** – Computed by the customizer.
- **Alternate Settings** – Computed by the customizer.
- **Class** – **HID**, **Vendor Specific**, or **Undefined**
- **Subclass** – Dependent on the selected class

- **Protocol**

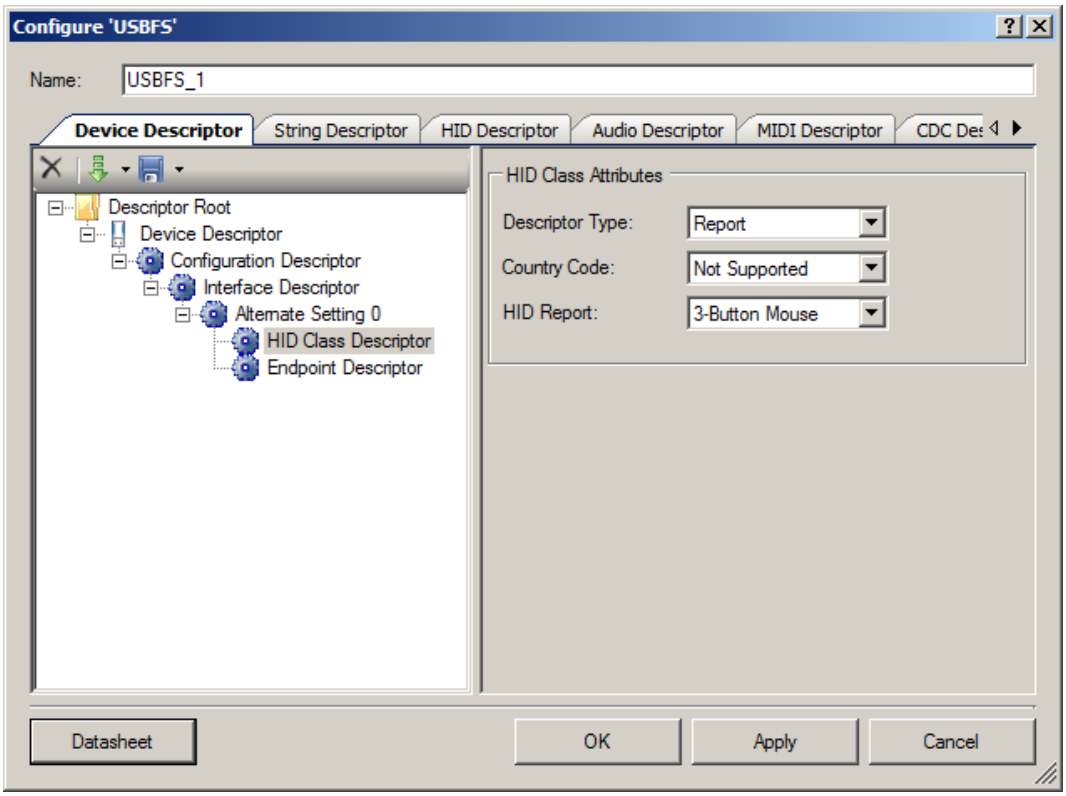
Note String descriptors are optional. If a device does not support string descriptors, all references to string descriptors within the device, configuration, and interface descriptors must be set to zero.

HID Class Descriptor

The HID Class Descriptor item does not display by default. It is used to add a HID Report to the Alternate Setting.

To Add HID Class Descriptor

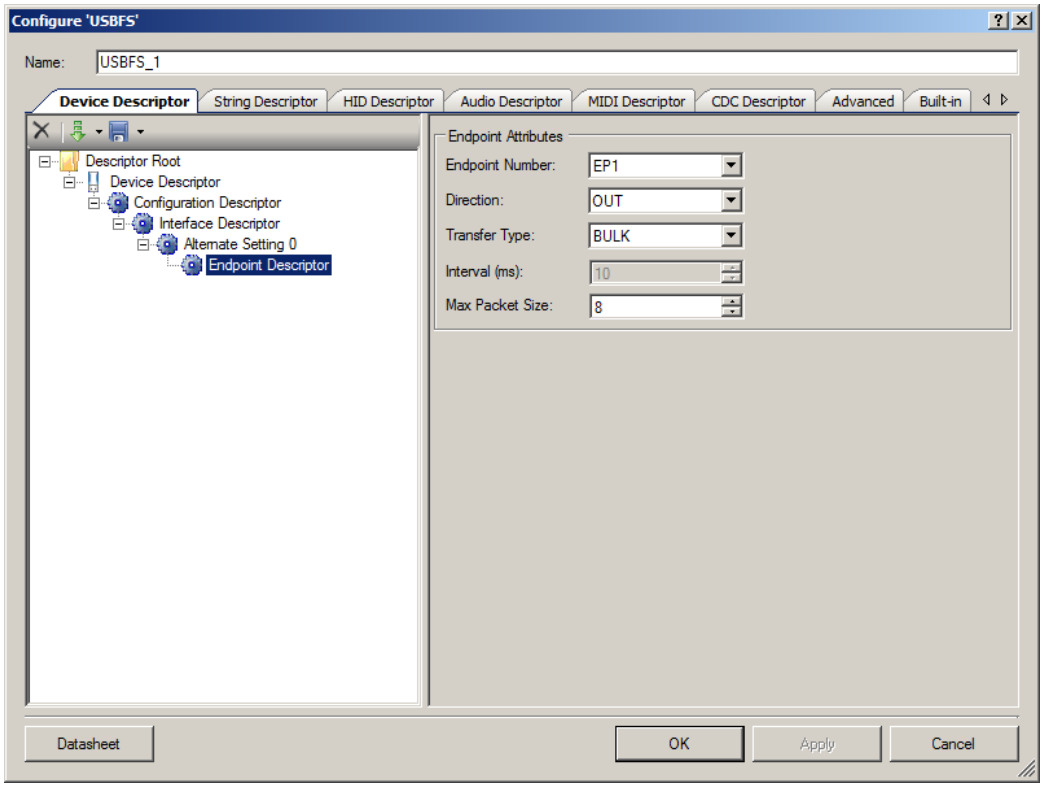
1. Select an **Alternate Setting** item in the **Descriptor Root** tree.
2. Under **Interface Attributes** on the right, select **HID** for the **Class** field.



HID Class Attributes

- **Descriptor Type** – Constant name identifying type of class descriptor
- **Country Code** – Numeric expression identifying country code of the localized hardware
- **HID Report** – List of available report descriptors. Report descriptors are taken from the **HID Descriptor** tab. This field is required.

Endpoint Descriptor



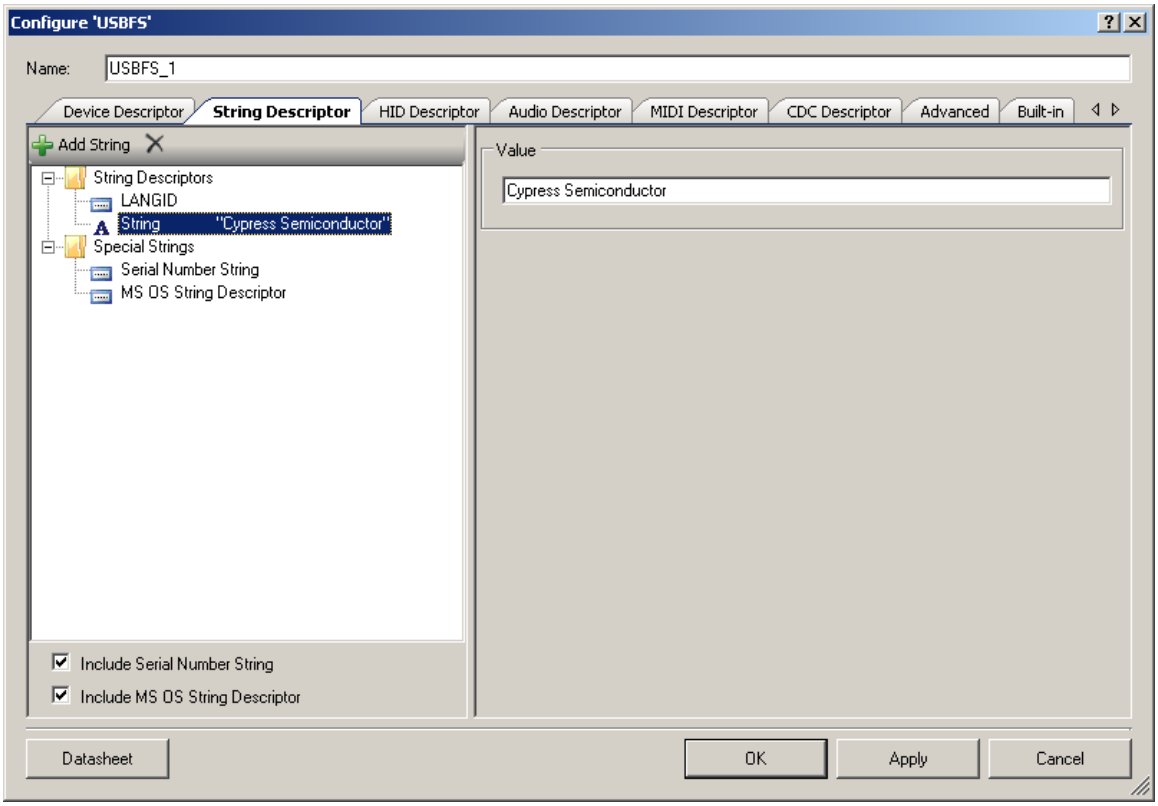
Endpoint Attributes

- **Endpoint Number**
- **Direction** – Input or Output. USB transfers are host centric; therefore, **IN** refers to transfers to the host; **OUT** refers to transfers from the host.
- **Transfer Type** – Control (**CONT**), Interrupt (**INT**), Bulk (**BULK**), or Isochronous Data (**ISOC**) transfers
- **Interval** (ms) – Polling interval specific to this endpoint. A full-speed endpoint can specify a period from 1 ms to 255 ms.
- **Max Packet Size** (bytes) – For a full-speed device the **Max Packet Size** is 64 bytes for bulk or interrupt endpoints and 512 (1023 for Automatic DMA mode) bytes for isochronous endpoints. For full-speed device bulk endpoints only 8-, 16-, 32-, and 64-byte values are allowed.

  The maximum packet size for the isochronous endpoints is limited by the local memory size in the Manual Memory Management mode of operation, while the DMA w/Automatic Memory Management mode of operation has no such limitation. This is because the local memory is treated as a temporary buffer.

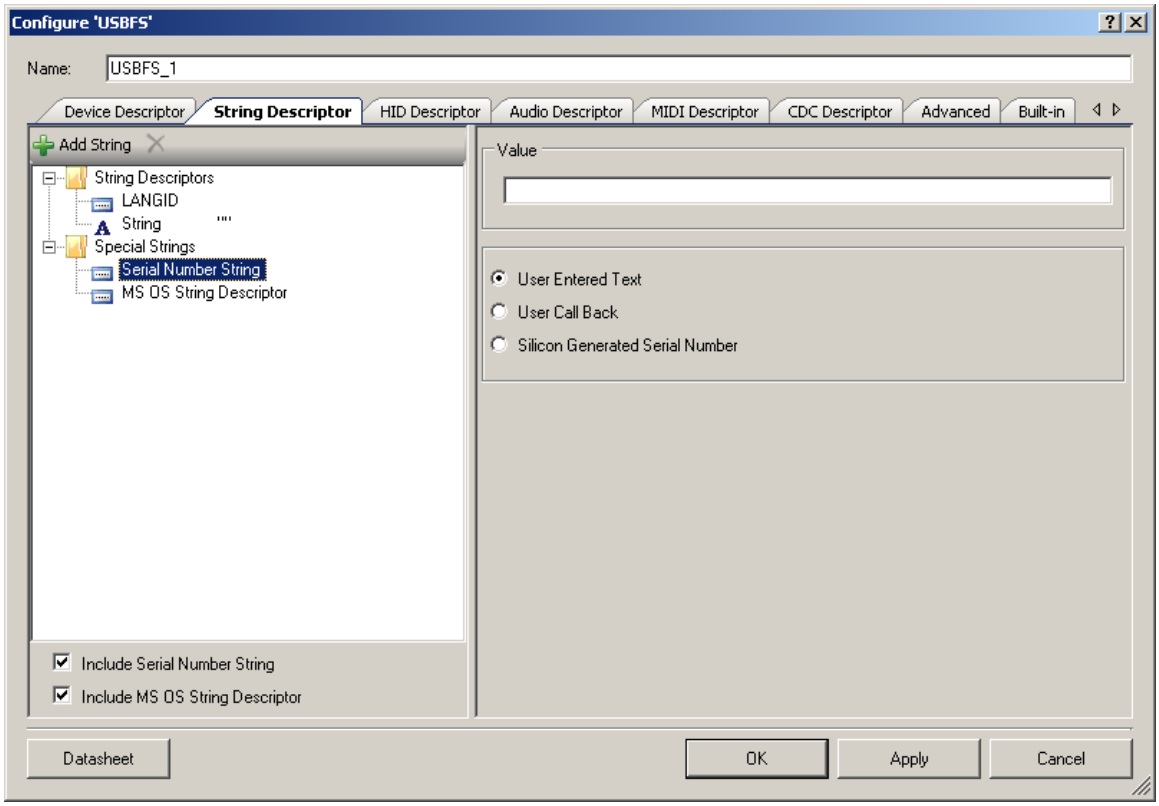
String Descriptor Tab

String Descriptors



- **LANGID** – Language ID selection
- **String** – Value of string descriptor

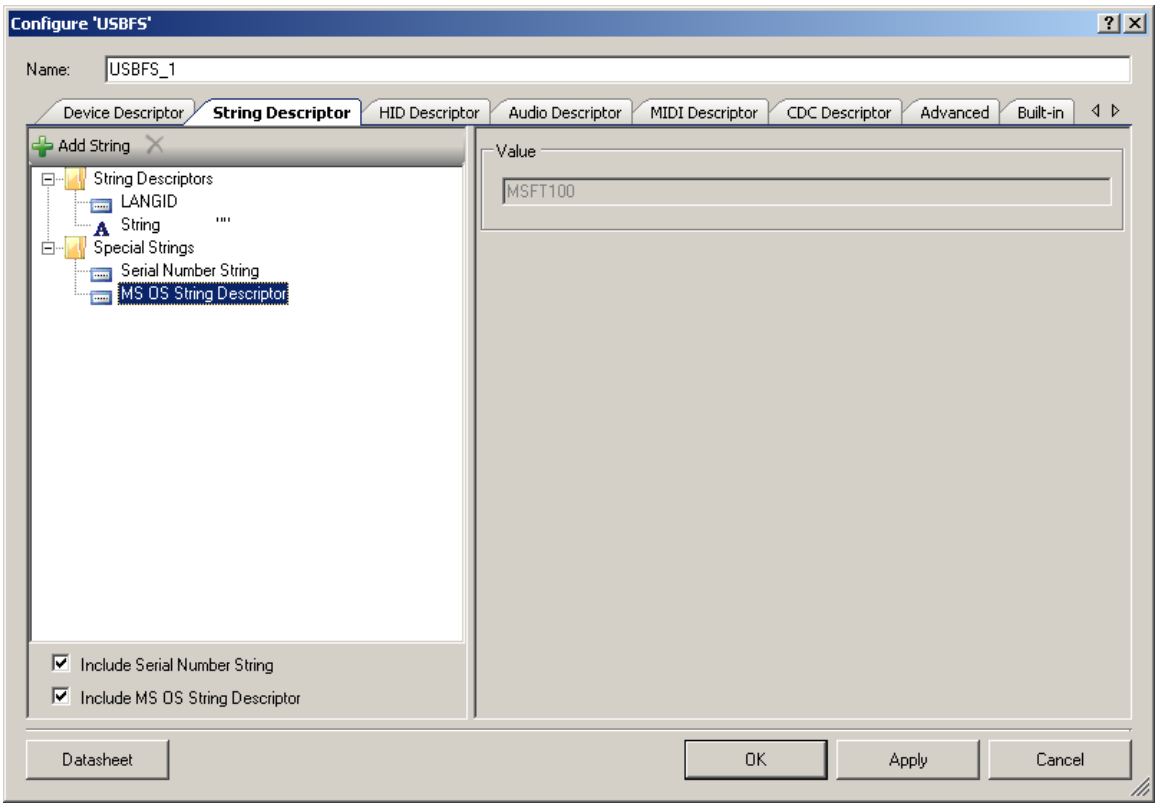
Serial Number String



- **Value** – Default string
- **User Entered Text** – Enables the **Value** text box
- **User Call Back** – The USBFS_SerialNumString() function sets the pointer to use the user-generated serial number string descriptor. The application firmware may supply the source of the USB device descriptor's serial number string during run time.
- **Silicon Generated Serial Number** – This number is applied to non-volatile memory in the device at manufacturing time. It is not guaranteed to be unique.

MS OS String Descriptor

Microsoft OS Descriptors provide a way for USB devices to supply additional configuration information to the latest Microsoft operating systems.

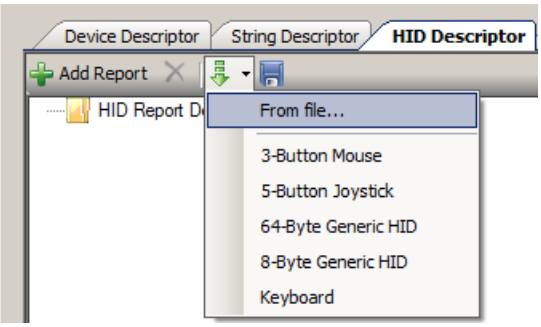


- **Value** – Constant string **MSFT100**

HID Descriptor Tab

The **HID Descriptor** tab allows you to quickly build HID descriptors for your device.

Toolbar Buttons



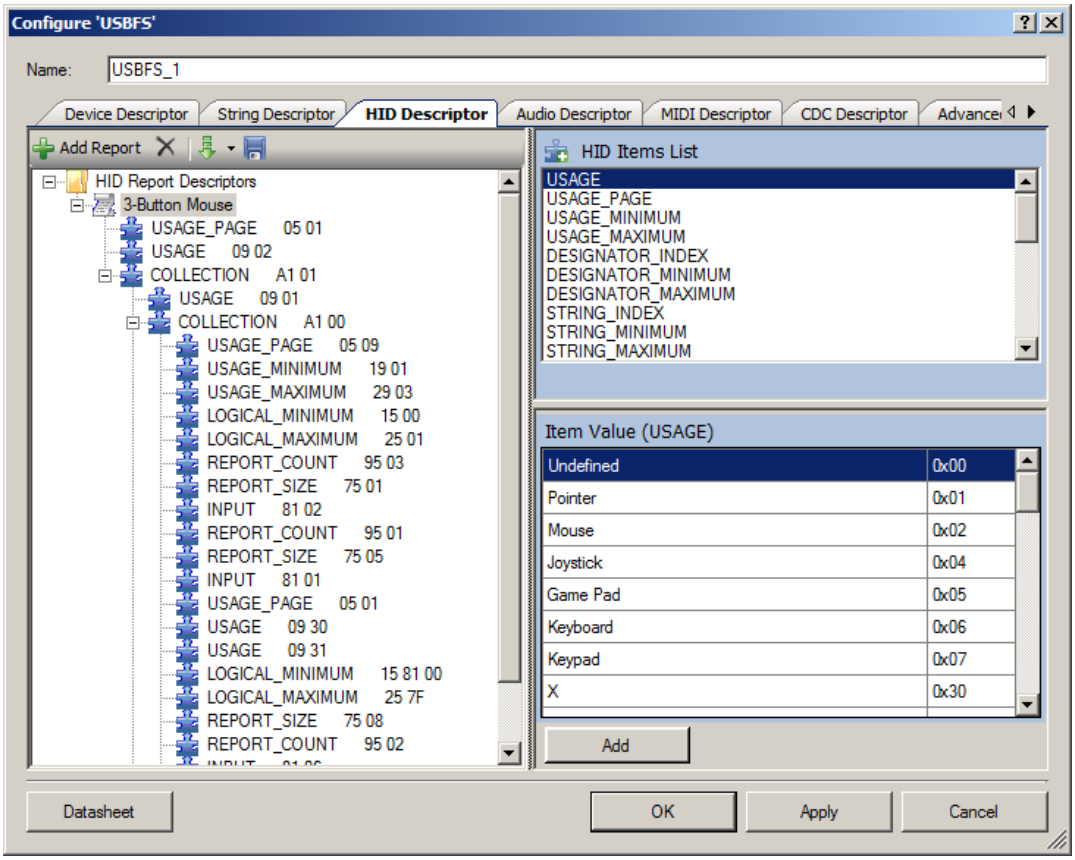
Use the **Add Report** button to add and configure HID Report Descriptors.

Use the **Import** button to import the HID report. In the drop-down list you can choose one of the templates or **From file**.

The template options immediately load the selected HID report. The **From file** option will open an HID report that was created by the USBFS component, or from the USB-IF HID Descriptor Tool. Refer to the USB-IF website for information about the HID Descriptor Tool: http://www.usb.org/developers/hidpage#HID Descriptor Tool.

Version 2.4 of the tool is supported. The file formats supported are .hid, .h, and .dat. You need to choose an appropriate file extension in the Open File dialog depending on the source file format.

HID Descriptors



- **HID Items List** – Items to add in the HID report
- **Item Value** – Value of the item that is selected either in **HID Items List** or in the tree

Audio Descriptor Tab

The **Audio Descriptor** tab is used to add and configure audio interface descriptors. See the USBFS Audio section for more information.

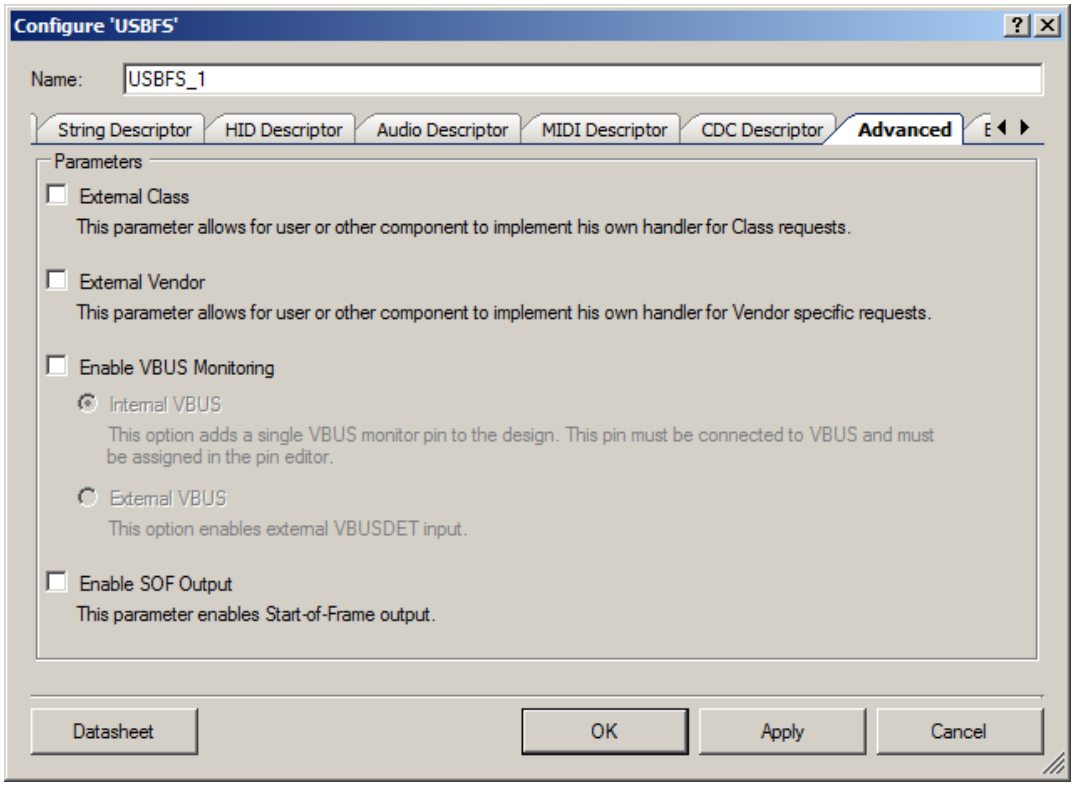
MIDI Descriptor Tab

The **MIDI Descriptor** tab is used to add and configure MIDI Streaming interface descriptors. See the USBFS MIDI section for more information.

CDC Descriptor Tab

The **CDC Descriptor** tab is used to add and configure communications and data interface descriptors. See the USBUART section for more information.

Advanced Tab



External Class

This parameter allows for the user firmware, or other components at the solutions level, to manage the class requests. The USBFS_DispatchClassRqst() function should be implemented if this parameter is enabled.

External Vendor

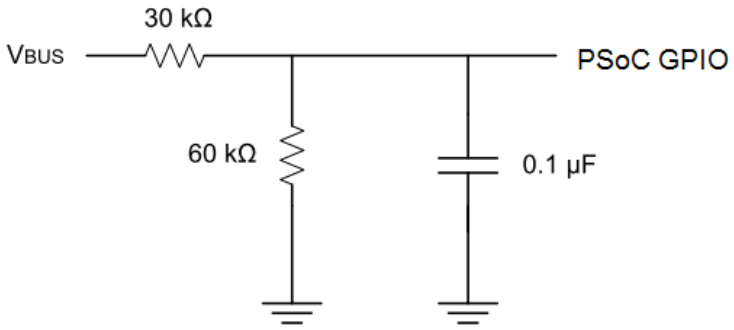
This parameter allows for the user firmware, or other components at the solutions level, to manage the vendor-specific requests. The USBFS_HandleVendorRqst() function should be implemented if this parameter is enabled.

Enable VBUS Monitoring

The USB specification requires that no device supplies current on VBUS at its upstream facing port at any time. To meet this requirement, the device must monitor for the presence or absence of VBUS and remove power from the D+/D– pull-up resistor if VBUS is absent.

For bus-powered designs, power will obviously be removed when the USB cable is removed from a host; however, for self-powered designs it is imperative for proper operation and USB certification that your device complies with this requirement.

This parameter adds a single VBUS monitor pin to the design if the **Internal VBUS** option is selected. By default, the drive mode of this pin is configured to High Impedance Digital, and could be set to a different mode by using the pin-specific API USBFS_VBUS_SetDriveMode(). When the **External VBUS** option is selected, the digital input Pin component should be placed on the schematic and must be connected to vbusdet input terminal. This pin must be connected to the VBUS through the resistive network and must be assigned in the Pin Editor. An example schematic is shown in the following figure.



The monitoring pin can also be directly connected to VBUS, if it is assigned to a SIO port in the PSoC Creator Pin Editor. This is due to the hot swap capabilities of the SIO.

The USBFS_VBusPresent() function returns the status of the VBUS. See the USB Compliance for Self-Powered Devices section for additional information.

Enable SOF Output

This parameter enables the Start-of-Frame output.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections discuss each function in more detail.

By default, PSoC Creator assigns the instance name “USBFS_1” to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “USBFS.”

Basic USBFS Device APIs

Function	Description
USBFS_Start()	Activates the component for use with the device and specific voltage mode.
USBFS_Init()	Initializes the component’s hardware.

Function	Description
USBFS_InitComponent()	Initializes the component's global variables and initiates communication with host by pull-up D+ line.
USBFS_Stop()	Disables the component.
USBFS_GetConfiguration()	Returns the currently assigned configuration. Returns 0 if the device is not configured.
USBFS_IsConfigurationChanged()	Returns the clear-on-read configuration state.
USBFS_GetInterfaceSetting()	Returns the current alternate setting for the specified interface.
USBFS_GetEPState()	Returns the current state of the specified USBFS endpoint.
USBFS_GetEPAckState()	Determines whether an ACK transaction occurred on this endpoint.
USBFS_GetEPCount()	Returns the current byte count from the specified USBFS endpoint.
USBFS_InitEP_DMA()	Initializes DMA for EP data transfers.
USBFS_LoadInEP()	Loads and enables the specified USBFS endpoint for an IN transfer.
USBFS_ReadOutEP()	Reads the specified number of bytes from the Endpoint RAM and places it in the RAM array pointed to by pSrc. Returns the number of bytes sent by the host.
USBFS_EnableOutEP()	Enables the specified USB endpoint to accept OUT transfers.
USBFS_DisableOutEP()	Disables the specified USB endpoint to NAK OUT transfers.
USBFS_SetPowerStatus()	Sets the device to self powered or bus powered.
USBFS_Force()	Forces a J, K, or SE0 State on the USB D+/D– pins. Normally used for remote wakeup.
USBFS_SerialNumString()	Provides the source of the USB device serial number string descriptor during run time.
USBFS_TerminateEP()	Terminates endpoint transfers.
USBFS_VBusPresent()	Determines VBUS presence for self-powered devices.

Global Variables

Variable	Description
USBFS_initVar	Indicates whether the USBFS has been initialized. The variable is initialized to 0 and set to 1 the first time USBFS_Start() is called. This allows the component to restart without reinitialization after the first call to the USBFS_Start() routine. If reinitialization of the component is required, the variable should be set to 0 before the USBFS_Start() routine is called. Alternatively, the USBFS can be reinitialized by calling the USBFS_Init() and USBFS_InitComponent() functions.
USBFS_device	Contains the started device number. This variable is set by the USBFS_Start() or USBFS_InitComponent() APIs.

Variable	Description
USBFS_transferState	This variable is used by the communication functions to handle the current transfer state. Initialized to TRANS_STATE_IDLE in the USBFS_InitComponent() API and after a complete transfer in the status stage. Changed to the TRANS_STATE_CONTROL_READ or TRANS_STATE_CONTROL_WRITE in setup transaction depending on the request type.
USBFS_configuration	Contains the current configuration number, which is set by the host using a SET_CONFIGURATION request. This variable is initialized to zero in USBFS_InitComponent() API and returned to the application level by the USBFS_GetConfiguration() API._
USBFS_configurationChanged	This variable is set to one after SET_CONFIGURATION and SET_INTERFACE requests. It is returned to the application level by the USBFS_IsConfigurationChanged() API.
USBFS_deviceStatus	This is a two-bit variable that contains power status in the first bit (DEVICE_STATUS_BUS_POWERED or DEVICE_STATUS_SELF_POWERED) and remote wakeup status (DEVICE_STATUS_REMOTE_WAKEUP) in the second bit. This variable is initialized to zero in USBFS_InitComponent() API, configured by the USBFS_SetPowerStatus() API.

void USBFS_Start(uint8 device, uint8 mode)

Description: This function performs all required initialization for the USBFS component.

Parameters: uint8 device: Contains the device number of the desired device descriptor. The device number can be found in the Device Descriptor Tab of Configure dialog, under the settings of desired Device Descriptor, in the **Device Number** field.

uint8 mode: Operating voltage. This determines whether the voltage regulator is enabled for 5-V operation or if pass-through mode is used for 3.3-V operation. Symbolic names and their associated values are given in the following table.

Power Setting	Notes
USBFS_3V_OPERATION	Disable the voltage regulator and pass-through V_{CC} for pull-up
USBFS_5V_OPERATION	Enable the voltage regulator and use the regulator for pull-up
USBFS_DWR_VDDD_OPERATION	Enable or disable the voltage regulator depending on V_{DDD} voltage configuration in DWR

Return Value: None

Side Effects: None

void USBFS_Init(void)

Description: This function initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call USBFS_Init() because the USBFS_Start() routine calls this function and is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: None

void USBFS_InitComponent(uint8 device, uint8 mode)

Description: This function initializes the component's global variables and initiates communication with the host by pull-up D+ line.

Parameters: uint8 device: Contains the device number of the desired device descriptor. The device number can be found in the Device Descriptor Tab of Configure dialog, under the settings of desired Device Descriptor, in the **Device Number** field.

uint8 mode: Operating voltage. This determines whether the voltage regulator is enabled for 5-V operation or if pass-through mode is used for 3.3-V operation. Symbolic names and their associated values are given in the following table.

Power Setting	Notes
USBFS_3V_OPERATION	Disable the voltage regulator and pass-through V_{CC} for pull-up
USBFS_5V_OPERATION	Enable the voltage regulator and use the regulator for pull-up
USBFS_DWR_VDDD_OPERATION	Enable or disable the voltage regulator depending on V_{DDD} voltage configuration in DWR

Return Value: None

Side Effects: None

void USBFS_Stop(void)

Description: This function performs all necessary shutdown tasks required for the USBFS component.

Parameters: None

Return Value: None

Side Effects: None

uint8 USBFS_GetConfiguration(void)

Description: This function gets the current configuration of the USB device.

Parameters: None

Return Value: uint8: Returns the currently assigned configuration. Returns 0 if the device is not configured.

Side Effects: None

uint8 USBFS_IsConfigurationChanged(void)

Description: This function returns the clear-on-read configuration state. It is useful when the PC sends double SET_CONFIGURATION requests with the same configuration number.

Parameters: None

Return Value: uint8: Returns a nonzero value when a new configuration has been changed; otherwise, it returns zero.

Side Effects: None

uint8 USBFS_GetInterfaceSetting(uint8 interfaceNumber)

Description: This function gets the current alternate setting for the specified interface.

Parameters: uint8 interfaceNumber: Interface number

Return Value: uint8: Returns the current alternate setting for the specified interface.

Side Effects: None

uint8 USBFS_GetEPState(uint8 epNumber)

Description: This function returns the state of the requested endpoint.

Parameters: uint8 epNumber: Data endpoint number

Return Value: uint8: Returns the current state of the specified USBFS endpoint. Symbolic names and their associated values are given in the following table. Use these constants whenever you write code to change the state of the endpoints, such as ISR code, to handle data sent or received.

Return Value	Description
USBFS_NO_EVENT_PENDING	The endpoint is awaiting SIE action
USBFS_EVENT_PENDING	The endpoint is awaiting CPU action
USBFS_NO_EVENT_ALLOWED	The endpoint is locked from access
USBFS_IN_BUFFER_FULL	The IN endpoint is loaded and the mode is set to ACK IN
USBFS_IN_BUFFER_EMPTY	An IN transaction occurred and more data can be loaded
USBFS_OUT_BUFFER_EMPTY	The OUT endpoint is set to ACK OUT and is waiting for data
USBFS_OUT_BUFFER_FULL	An OUT transaction has occurred and data can be read

Side Effects: None

uint8 USBFS_GetEPAckState(uint8 epNumber)

Description: This function determines whether an ACK transaction occurred on this endpoint by reading the ACK bit in the control register of the endpoint. It does not clear the ACK bit.

Parameters: uint8 epNumber: Contains the data endpoint number.

Return Value: uint8: If an ACKed transaction occurred, this function returns a nonzero value. Otherwise, it returns zero.

Side Effects: None

uint16 USBFS_GetEPCount(uint8 epNumber)

Description: This function returns the transfer count for the requested endpoint. The value from the count registers includes two counts for the two-byte checksum of the packet. This function subtracts the two counts.

Parameters: uint8 epNumber: Contains the data endpoint number.

Return Value: uint16: Returns the current byte count from the specified USBFS endpoint or 0 for an invalid endpoint.

Side Effects: None

void USBFS_InitEP_DMA(uint8 epNumber, const uint8 *pData)

Description: This function allocates and initializes a DMA channel to be used by the USBFS_LoadInEP() or USBFS_ReadOutEP() APIs for data transfer. It is available when the Endpoint Memory Management parameter is set to DMA.

This function is automatically called from the USBFS_LoadInEP() and USBFS_ReadOutEP() APIs.

Parameters: uint8 epNumber: Contains the data endpoint number.

const uint8 *pData: Pointer to a data array that is related to the EP transfers.

Return Value: None

Side Effects: None

void USBFS_LoadInEP(uint8 epNumber, const uint8 pData[], uint16 length)

Description: Manual mode: This function loads and enables the specified USB data endpoint for an IN data transfer.

Manual DMA:

- Configures DMA for a data transfer from data RAM to endpoint RAM.
- Generates request for a transfer.

Automatic DMA:

- Configures DMA. This is required only once, so it is done only when parameter pData is not NULL. When the pData pointer is NULL, the function skips this task.
- Sets Data ready status: This generates the first DMA transfer and prepares data in endpoint RAM memory. Set the pData parameter to NULL to execute this task.

Parameters: uint8 epNumber: Contains the data endpoint number.

const uint8 pData[]: Pointer to a data array from which the data for the endpoint space is loaded.

uint16 length: The number of bytes to transfer from the array and then send as a result of an IN request. Valid values are between 0 and 512 (1023 for Automatic DMA mode).

Return Value: None

Side Effects: None

uint16 USBFS_ReadOutEP(uint8 epNumber, uint8 pData[], uint16 length)

Description: Manual mode: This function 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 or the number of bytes requested by the wCount parameter.

Manual DMA:

- Configure DMA for a transfer data from endpoint RAM to data RAM.
- Generate request for a transfer.
- After the USB_ReadOutEP() API and before the expected data use it must wait for the DMA transfer to complete. For example, by checking EPstate:

```
while (USBFS_GetEPState(OUT_EP) == USB_OUT_BUFFER_FULL);
```

Automatic DMA:

- Configure DMA. This is required only once.

Parameters: uint8 epNumber: Contains the data endpoint number.

uint8 pData[]: Pointer to a data array to which the data from the endpoint space is loaded.

uint16 length: The number of bytes to transfer from the USB OUT endpoint and load into data array. Valid values are between 0 and 512 (1023 for Automatic DMA mode). The function moves fewer than the requested number of bytes if the host sends fewer bytes than requested.

Return Value: uint16: Number of bytes received

Side Effects: None

void USBFS_EnableOutEP(uint8 epNumber)

Description: This function enables the specified endpoint for OUT transfers. Do not call this function for IN endpoints.

Parameters: uint8 epNumber: Contains the data endpoint number.

Return Value: None

Side Effects: None

void USBFS_DisableOutEP(uint8 epNumber)

Description: This function disables the specified USBFS OUT endpoint. Do not call this function for IN endpoints.

Parameters: uint8 epNumber: Contains the data endpoint number.

Return Value: None

Side Effects: None

void USBFS_SetPowerStatus(uint8 powerStatus)

Description: This function sets the current power status. 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 can 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. You should call this function any time your device changes from self powered to bus powered or vice versa, and set the status appropriately.

Parameters: uint8 powerStatus: Contains the desired power status, one for self powered or zero for bus powered. Symbolic names and their associated values are given here:

Power Status	Description
USBFS_DEVICE_STATUS_BUS_POWERED	Set the device to bus powered
USBFS_DEVICE_STATUS_SELF_POWERED	Set the device to self powered

Return Value: None

Side Effects: None

void USBFS_Force(uint8 state)

Description: This function forces a USB J, K, or SE0 state on the D+/D– lines. It provides the necessary mechanism for a USB device application to perform a USB Remote Wakeup. For more information, see the USB 2.0 Specification for details on Suspend and Resume.

Parameters: uint8 state: A byte indicating which of the four bus states to enable. Symbolic names and their associated values are listed here:

State	Description
USBFS_FORCE_SE0	Force a Single Ended 0 onto the D+/D– lines
USBFS_FORCE_J	Force a J State onto the D+/D– lines
USBFS_FORCE_K	Force a K State onto the D+/D– lines
USBFS_FORCE_NONE	Return bus to SIE control

Return Value: None

Side Effects: None

void USBFS_SerialNumString(uint8 snString[])

Description: This function is available only when the **User Call Back** option in the **Serial Number String** descriptor properties is selected. Application firmware can provide the source of the USB device serial number string descriptor during run time. The default string is used if the application firmware does not use this function or sets the wrong string descriptor.

Parameters: uint8 snString[]: Pointer to the user-defined string descriptor. The string descriptor should meet the *Universal Serial Bus Specification revision 2.0* chapter 9.6.7

Offset	Size	Value	Description
0	1	N	Size of this descriptor in bytes
1	1	0x03	STRING Descriptor Type
2	N - 2	Number	UNICODE encoded string

For example: *uint8 snString[16]={0x0E,0x03,'F',0,'W',0,'S',0,'N',0,'0',0,'1',0};*

Return Value: None

Side Effects: None

void USBFS_TerminateEP(uint8 epNumber)

Description: This function terminates the specified USBFS endpoint. This function should be used before endpoint reconfiguration.

Parameters: uint8 epNumber: Contains the data endpoint number.

Return Value: None

Side Effects: The device responds with a NAK for any transactions on the selected endpoint.

uint8 USBFS_VBusPresent(void)

Description: Determines VBUS presence for self-powered devices.

This function is available when the VBUS Monitoring option is enabled in the Advanced tab.

Parameters: None

Return Value: The return value can be the following:

Return Value	Description
1	VBUS is present
0	VBUS is absent

Side Effects: None

Human Interface Device (HID) Class Support

Function	Description
USBFS_UpdateHIDTimer()	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.
USBFS_GetProtocol()	Returns the protocol for the specified interface

Global Variables

If your HID descriptor contains reports, the Customizer creates a 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 HID Report descriptor as well as for multiple Report IDs per report type.

Variable	Description
USBFS_hidProtocol	This variable is initialized in the USBFS_InitComponent() API to the PROTOCOL_REPORT value. It is controlled by the host using the HID_SET_PROTOCOL request. The value is returned to the user code by the USBFS_GetProtocol() API.
USBFS_hidIdleRate	This variable controls the HID report rate. It is controlled by the host using the HID_SET_IDLE request and used by the USBFS_UpdateHIDTimer() API to reload timer.
USBFS_hidIdleTimer	This variable contains the timer counter, which is decremented and reloaded by the USBFS_UpdateHIDTimer() API.
USBFS_DEVICEx_CONFIGURATIONx_INTERFACEx_ALTERNATEx_HID_FEATURE[1]_BUF_IDx[2] [][3]	This optional report buffer is variable when feature (in or out) report descriptor is created inside the HID descriptor. The size of this buffer is automatically calculated by the customizer based on the HID report definition and defined as USBFS_DEVICEx_CONFIGURATIONx_INTERFACEx_ALTERNATEx_HID_FEATURE[1]_BUF_SIZE_IDx[2][3]. The Host controls this buffer by using SET_REPORT and GET_REPORT requests. The user code can check the status completion block to determine if a control transfer on the specific report ID is complete or not.

[1]. The “FEATURE” field in the variable name represents the report type and will be changed to “IN” or “OUT” for the IN or OUT reports.

[2]. The “_IDx” field in the variable name is present only when report ID is specified in the report descriptor and the “x” symbol will be changed to the associated report ID number.

[3]. The “x” symbol in the name depends on the associated device, configuration, interface and alternate setting number taken from Descriptor Root in the customizer.

<table>
<tr><th>Variable</th><th>Description</th></tr>
<tr><td>USBFS_DEVICEx_CONFIGURATIONx_INTERFACEx_ALTERNATEx_HID_FEATURE[1]_RPT_SCB_IDx[2][3]</td><td>The status completion block contains two data items, a one byte completion status code and a two byte transfer length. This block has the following structure.
<pre>typedef struct _USBFS_XferStatusBlock
{
    uint8 status;
    uint16 length;
} T_USBFS_XFER_STATUS_BLOCK;</pre>
The “main” application monitors the completion status to determine how to proceed. Completion status codes are found in the following table. The transfer length is the actual number of data bytes transferred.
<table>
<tr><th>Return Value</th><th>Notes</th></tr>
<tr><td>USBFS_XFER_IDLE</td><td>Indicates that the associated data buffer does not have valid data and the application should not use the buffer. The actual data transfer takes place while the completion code is USBFS_XFER_IDLE, although it does not indicate a transfer is in progress.</td></tr>
<tr><td>USBFS_XFER_STATUS_ACK</td><td>Indicates the control transfer status stage completed successfully. At this time, the application uses the associated data buffer and its contents.</td></tr>
<tr><td>USBFS_XFER_PREMATURE</td><td>Indicates that the control transfer was interrupted by the SETUP of a subsequent control transfer. For control writes, the content of the associated data buffer contains the data up to the premature completion.</td></tr>
<tr><td>USBFS_XFER_ERROR</td><td>Indicates that the expected status stage token was not received.</td></tr>
</table></td></tr>
</table>

uint8 USBFS_UpdateHIDTimer(uint8 interface)

Description: This function updates the HID Report idle timer and returns the status and reloads the timer if it expires.

Parameters: uint8 interface: Contains the interface number.

Return Value: uint8: Returns the state of the HID timer. Symbolic names and their associated values are given here:

Return Value	Notes
USBFS_IDLE_TIMER_EXPIRED	The timer expired.
USBFS_IDLE_TIMER_RUNNING	The timer is running.
USBFS_IDLE_TIMER_IDEFINITE	The report is sent when data or state changes.

Side Effects: None

uint8 USBFS_GetProtocol(uint8 interface)

Description: This function returns the HID protocol value for the selected interface.

Parameters: uint8 interface: Contains the interface number.

Return Value: uint8: Returns the protocol value.

Side Effects: None

Bootloader Support

The USBFS component can be used as a communication component for the Bootloader. You should use the following configurations to support communication protocol from an external system to the Bootloader:

- Endpoint Number: EP1, Direction: OUT, Transfer Type: INT, Max Packet Size: 64
- Endpoint Number: EP2, Direction: IN, Transfer Type: INT, Max Packet Size: 64

Full recommended configurations are stored in the template file (*bootloader.root.xml*). Select **Descriptor Root** on the **Device Descriptor** tree, click the **Import** button, browse to the following directory, and open the *bootloader.root.xml* file.

<INSTALL>\psoc\content\cycomponentlibrary\CyComponentLibrary.cylib\USBFS_v2.80\Custom\template

See the *System Reference Guide* for more information about the Bootloader.

The USBFS component provides a set of API functions for Bootloader use.

Function	Description
USBFS_CyBtldrCommStart()	Performs all required initialization for the USBFS component, waits on enumeration, and enables communication.
USBFS_CyBtldrCommStop()	Calls the USBFS_Stop() function.
USBFS_CyBtldrCommReset()	Resets the receive and transmit communication buffers.
USBFS_CyBtldrCommWrite()	Allows the caller to write data to the bootloader host. The function handles polling to allow a block of data to be completely sent to the host device.
USBFS_CyBtldrCommRead()	Allows the caller to read data from the bootloader host. The function handles polling to allow a block of data to be completely received from the host device.

void USBFS_CyBtldrCommStart(void)

Description: This function performs all required initialization for the USBFS component, waits on enumeration, and enables communication.

Parameters: None

Return Value: None

Side Effects: This function starts the USBFS with 3-V operation.

void USBFS_CyBtldrCommStop(void)

Description: This function performs all necessary shutdown tasks required for the USBFS component.

Parameters: None

Return Value: None

Side Effects: Calls the USBFS_Stop() function.

void USBFS_CyBtldrCommReset(void)

Description: This function resets receive and transmit communication buffers.

Parameters: None

Return Value: None

Side Effects: None

cystatus USBFS_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 *count, uint8 timeOut)

Description: This function allows the caller to write data to the bootloader host. It handles polling to allow a block of data to be completely sent to the host device.

Parameters: const uint8 pData[]: Pointer to the block of data to send to the device.

uint16 size: Number of bytes to write.

uint16 *count: Pointer to an unsigned short variable to write the number of bytes actually written.

uint8 timeout: Number of units to wait before returning because of a timeout.

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, see the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

cystatus USBFS_CyBtldrCommRead(uint8 pData[], uint16 size, uint16 *count, uint8 timeOut)

Description: This function allows the caller to read data from the bootloader host. It handles polling to allow a block of data to be completely received from the host device.

Parameters: uint8 pData[]: Pointer to the area to store the block of data received from the device.

uint16 size: Number of bytes to read.

uint16 *count: Pointer to an unsigned short variable to write the number of bytes actually read.

uint8 timeOut: Number of units to wait before returning because of a timeout.

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, see the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

USB Suspend, Resume, and Remote Wakeup

The USBFS component supports USB Suspend, Resume, and Remote Wakeup. Because these features are tightly coupled into the user application, the USBFS component provides a set of API functions.

Function	Description
USBFS_CheckActivity()	Checks and clears the USB bus activity flag. Returns 1 if the USB was active since the last check, otherwise returns 0.
USBFS_Suspend()	Disables the USBFS block and prepares for power down mode.
USBFS_Resume()	Enables the USBFS block after power down mode.
USBFS_RWUEnabled()	Returns current remote wakeup status.

uint8 USBFS_CheckActivity(void)

Description: This function returns the activity status of the bus and clears the status hardware to provide fresh activity status on the next call of this routine.

This function provides a means to determine whether any USB bus activity occurred. The application uses the function to determine if the conditions to enter USB Suspend were met.

Parameters: None

Return Value: uint8 cystatus: Standard API return values.

Return Value	Description
1	Bus activity was detected since the last call to this function
0	Bus activity was not detected since the last call to this function

Side Effects: None

void USBFS_Suspend(void)

Description: This function disables the USBFS block and prepares for power down mode. It should be called just before entering sleep.

USBFS_Suspend() also initializes the interrupt for the D+ pin for wakeup from the sleep mode from the PICU source.

After the conditions to enter USB suspend are met, the application takes appropriate steps to reduce current consumption to meet suspend current requirements. To put the USB SIE and transceiver into power down mode, the application calls the USBFS_Suspend() API function and the USBFS_CheckActivity() API to detect USB activity. This function disables the USBFS block, but maintains the current USB address (in the USBCR register). The device uses the sleep feature to reduce power consumption.

Parameters: void

Return Value: void

Side Effects: None

void USBFS_Resume(void)

Description: This function enables the USBFS block after power down mode. It should be called just after waking from sleep.

While the device is suspended, it periodically checks to determine if the conditions to leave the suspended state were met. One way to check resume conditions is to use the sleep timer to periodically wake the device. The second way is to configure the device to wake up from the PICU.

If the resume conditions are met, the application calls the USBFS_Resume() API function. This function enables the USBFS SIE and Transceiver, bringing them out of power down mode. It does not change the USB address field of the USBCR register; it maintains the USB address previously assigned by the host.

Parameters: void

Return Value: void

Side Effects: None

uint8 USBFS_RWUEnabled(void)

Description: This function returns the current remote wakeup status.

If the device supports remote wakeup, the application can use this function to determine if the host enabled remote wakeup. When the device is suspended and it determines the conditions to initiate a remote wakeup are met, the application uses the USBFS_Force() API function to force the appropriate J and K states onto the USB, signaling a remote wakeup.

Parameters: void

Return Value: True: Remote wakeup enabled

False: Remote wakeup disabled

Side Effects: None

Sleep mode API usage example, a PICU source is used for wakeup:

```
USBFS_Suspend();
CyPmSaveClocks();
CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_PICU);
CyPmRestoreClocks();
USBFS_Resume();
```

Audio Class Support

See the USBFS Audio section for information.

MIDI Class Support

See the USBFS MIDI section for information.

CDC Class Support

See the USBUART section for information.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The USBFS component has the following specific deviation:

Rule	Rule Class	Rule Description	Description of Deviation(s)
11.4	Advisory	A cast should not be performed between a pointer to object type and a different pointer to object type.	PutString() API has a pointer to string as an argument. This function converts this pointer to uint8 array to pass it to LoadInEP() API for transfer to HOST. Device Descriptor structures use pointer to void to combine different descriptor types. These pointers are cast to a different pointer type. Comment: This operation is safe because functions which parse the structures known the type of the object.
11.5	Required	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	Device Descriptor structures use const qualification. The const qualification is removed before passing the pointer to universal LoadInEP() API.
16.7	Advisory	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	This deviation only applies in DMA Memory management mode. pData argument of the ReadOutEP() API is used to modify the addressed object in Manual Memory management mode.

Rule	Rule Class	Rule Description	Description of Deviation(s)
17.4	Required	Array indexing shall be the only allowed form of pointer arithmetic.	The component applies array subscripting to an object of pointer type to access structures with descriptors.

This component has the following embedded components: Interrupt, Clock, DMA. Refer to the corresponding component datasheet for information on their MISRA compliance and specific deviations.

API Memory Usage

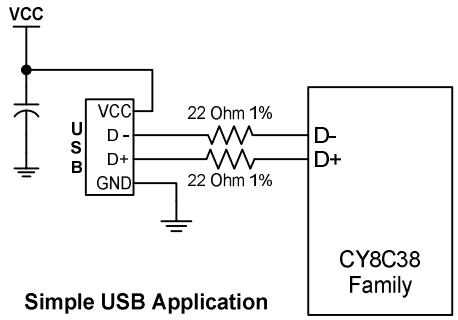
The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with the associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Default USBFS	8542	222	5795	242
Maximum, with HID,CDC, Audio, MIDI and Bootloader	19938	440	10866	468

Functional Description

The following diagram shows a simple bus-powered USB application with the D+ and D– pins from the PSoC device.



Simple USB Application

USB Compliance

USB drivers can present various bus conditions to the device, including Bus Resets, and different timing requirements. Not all of these can be correctly illustrated in the examples provided. It is your responsibility to design applications that conform to the USB spec.

USB Compliance for Self-Powered Devices

If the device that you are creating is self powered, you must connect a GPIO pin to VBUS through a resistive network and write firmware to monitor the status of the GPIO. You can use the USBFS_Start() and USBFS_Stop() API routines to control the D+ and D– pin pull-ups. The pull-up resistor does not supply power to the data line until you call USBFS_Start(). USBFS_Stop() disconnects the pull-up resistor from the data pin.

The device responds to GET_STATUS requests based on the status set with the USBFS_SetPowerStatus() function. To set the correct status, USBFS_SetPowerStatus() should be called at least once if your device is configured as self powered. You should also call the USBFS_SetPowerStatus() function any time your device changes status.

USB Standard Device Requests

This section describes the requests supported by the USBFS component. If a request is not supported, the USBFS component responds with a STALL, indicating a request error.

Standard Device Request	USB Component Support Description	USB 2.0 Spec Section
CLEAR_FEATURE	Device	9.4.1
	Interface	
	Endpoint	
GET_CONFIGURATION	Returns the current device configuration value	9.4.2

Standard Device Request	USB Component Support Description	USB 2.0 Spec Section
GET_DESCRIPTOR	Returns the specified descriptor if the descriptor exists.	9.4.3
GET_INTERFACE	Returns the selected alternate interface setting for the specified interface	9.4.4
GET_STATUS	Device	9.4.5
	Interface	
	Endpoint	
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 component parameter. TEST_MODE is not supported.	9.4.9
	Interface	
	Endpoint: The specified Endpoint is halted.	
SET_INTERFACE	Allows the host to select an alternate setting for the specified interface.	9.4.10
SYNCH_FRAME	Not supported. Future implementations of the component will add support to this request to enable Isochronous transfers with repeating frame patterns.	9.4.11

HID Class Request

Class Request	USBFS Component 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

AUDIO Class Request

See the Audio Class Request section under USBFS Audio for information.

CDC Class Request

See the CDC Class Request section under USBUART for information.

Interrupt Service Routine

Empty SOF ISR is provided with this component. It is disabled by default. If your application requires this interrupt it can be enabled by calling:

```
CyIntEnable(USBFS_SOF_VECT_NUM);
```

You can place custom code in the designated areas to perform whatever additional function is required.

In the **DMA with Manual and Automatic Memory Management** mode, the Arbiter interrupt (USBFS_arb_int) indicates the completion of service of a DMA request. It is critical for the system to set the priority of this interrupt higher than the priority of the USBFS_ep_[0..8] and USBFS_ord_int interrupts. Therefore, the priority for this interrupt is set to the USBFS_ARB_PRIOR value in the USBFS_Init() function.

Clock Selection

The USB hardware block requires that system clocks be configured through the PSoC Creator Design-Wide Resources Clock Editor. Clock settings have the following requirements when using the USBFS component:

- The USB Clock must be enabled.
- The ILO must be set to 100 kHz.

There are different ways to configure the system clocks to comply with these requirements. Figure 1 and Figure 2 show the set of options you may use. Your design may require different settings.

Figure 1. System Clock Configuration, IMO is a source clock

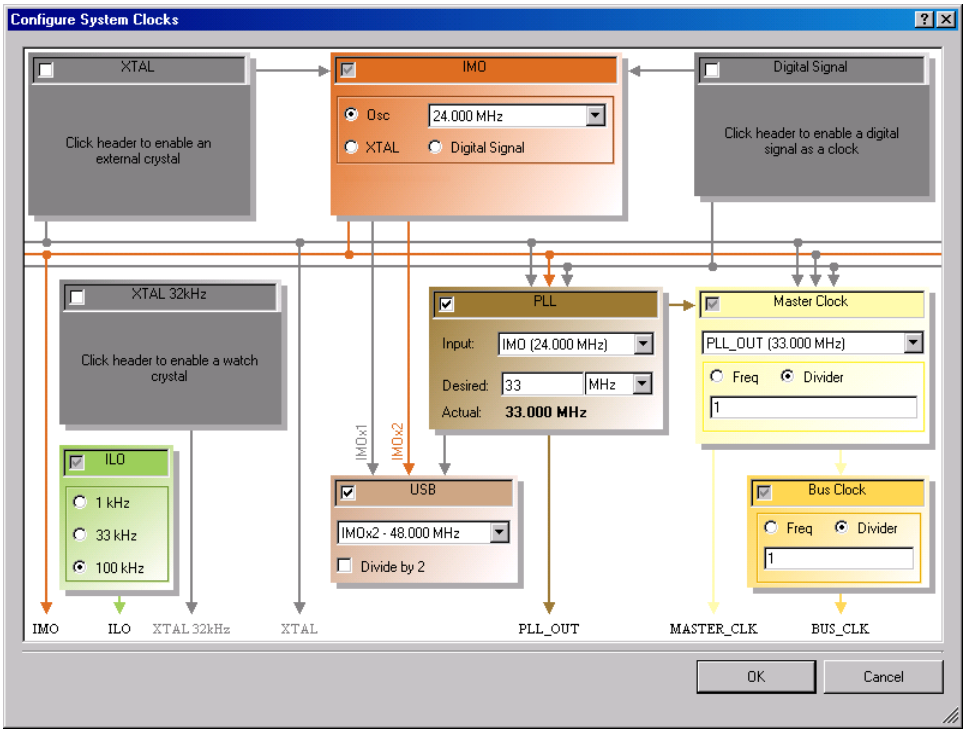
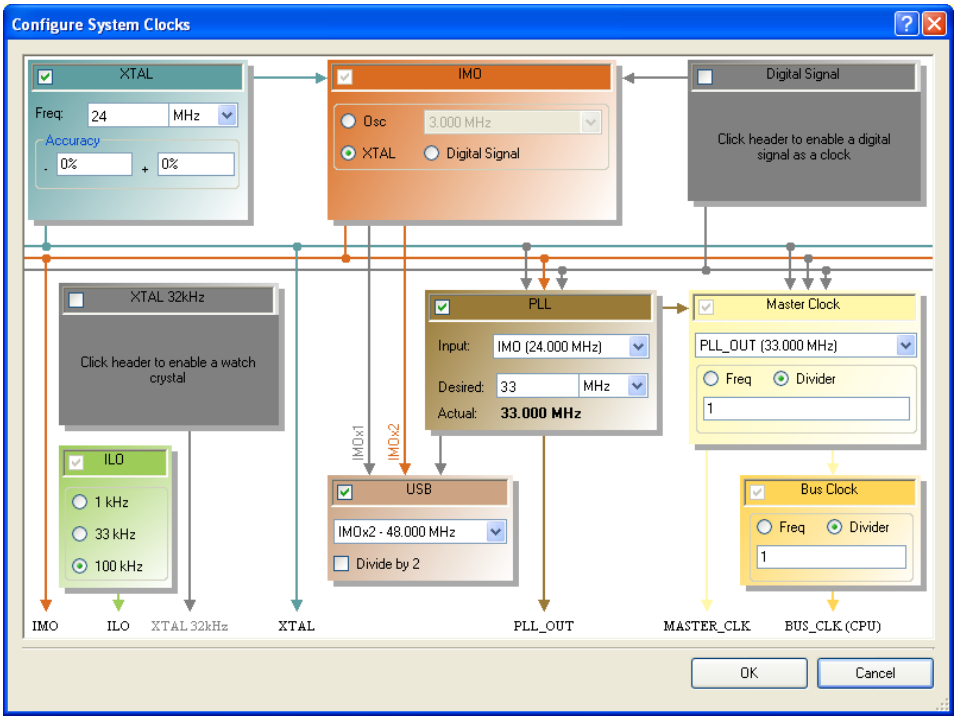


Figure 2. System Clock Configuration, XTAL is a clock source



USBFS Configurations

In addition to the base configuration, the USBFS component can be configured into the following options:

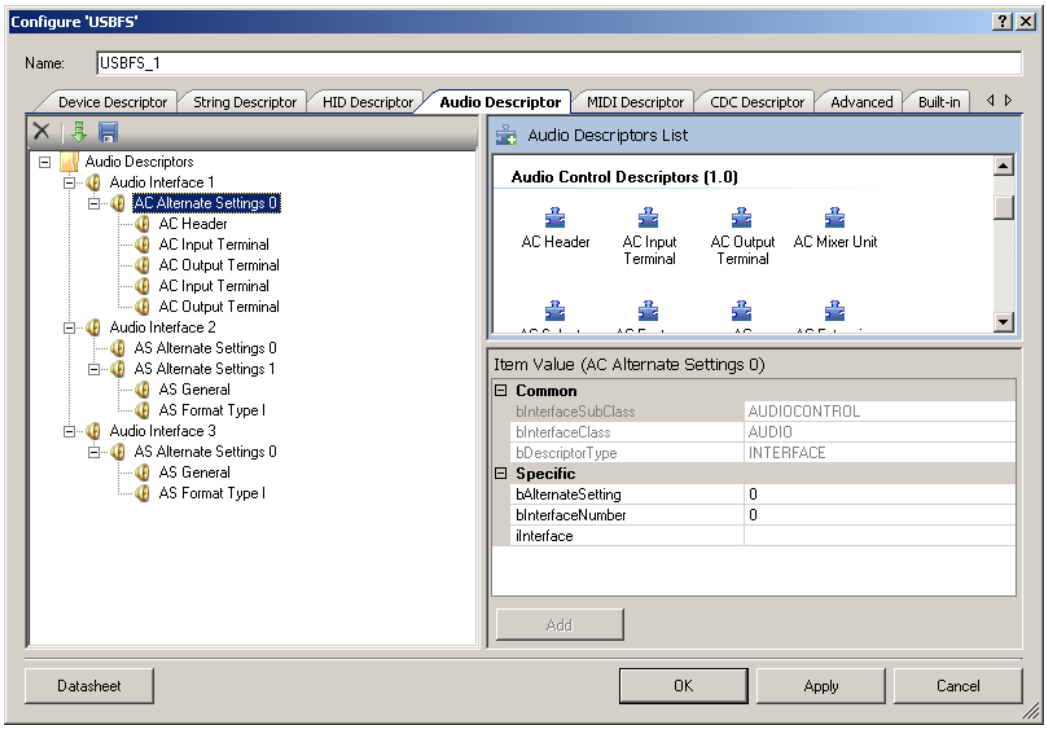
- USBFS Audio
- USBFS MIDI
- USBUART (CDC)

USBFS Audio

The USBFS component provides support for Audio class descriptors. The USBFS Audio interface is implemented according to the *Universal Serial Bus Device Class Definition for Audio Devices 1.0 and 2.0* specifications.

USBFS Audio Parameters

To add and configure audio interface descriptors, open the Configure USBFS dialog and click the **Audio Descriptor** tab.



To Add Audio Descriptors

1. Select the **Audio Descriptors** root item in the tree on the left.

2. Under the **Audio Descriptors List** on the right, select either the **Audio Control** or **Audio Streaming** interface.
3. Under **Item Value**, enter **bAlternateSetting** and **bInterfaceNumber** values as appropriate. Other fields are optional.

   Note These values are set manually. By contrast, for the general interface descriptors, these values are set automatically.
4. Click **Add** to add the descriptor to the tree on the left.

   You can rename the **Audio Interface x** title by selecting a node and clicking on it.

To Add Class-Specific Audio Control or Audio Streaming Interface Descriptors

1. Select the appropriate **AC Alternate Settings x** or **AS Alternate Settings x** item in the tree on the left.
2. Under the **Audio Descriptors List** on the right, select one of the items under **Audio Control Descriptors (1.0)**, **Audio Control Descriptors (2.0)**, **Audio Streaming Descriptors (1.0)**, or **Audio Streaming Descriptors (2.0)** as appropriate.

   Versions 1.0 and 2.0 refer to the versions of the corresponding specification document *Universal Serial Bus Device Class Definition for Audio Devices*.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add Audio Endpoint Descriptors

1. Select the appropriate **AC Alternate Settings x** or **AS Alternate Settings x** item in the tree on the left.
2. Under the **Audio Descriptors List** on the right, select the **Endpoint Descriptor** item.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

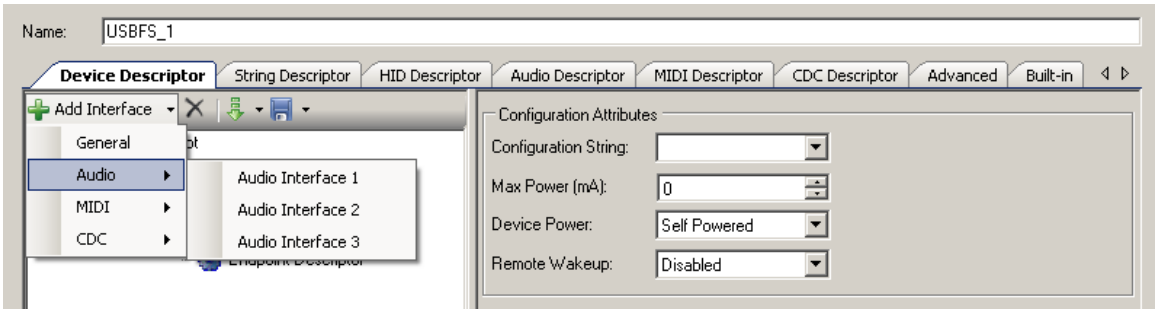
To Add Standard AS Isochronous Synch Endpoint Descriptor

1. Select the appropriate **Endpoint Descriptor** in the tree on the left.
2. Under the **Audio Descriptors List** on the right, select **AS Endpoint Descriptor**.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add the Configured Audio Interface Descriptor to the Device Descriptor Tree

1. Go to the **Device Descriptor** tab.
2. Select the **Configuration Descriptor** to which a new interface will belong.

3. Click the **Add Interface** tool button, choose **Audio**, and select the appropriate item to add.



Audio interfaces are disabled in the **Device Descriptor** tab list because they can only be edited on the **Audio Descriptor** tab.

Note Click **Apply** or **OK** to save the changes on the various tabs. If you click **Cancel**, the descriptors you added will not be saved.

USBFS Audio API Support

Global Variables

Variable	Description
USBFS_currentSampleFrequency	Contains the current audio sample frequency. It is set by the host using a SET_CUR request to the endpoint.

USBFS_frequencyChanged	Used as a flag for the user code, to inform it that the host has been sent a request to change the sample frequency. The sample frequency will be sent on the next OUT transaction. It contains the endpoint address when set. The following code is recommended for detecting new sample frequency in main code:
	`if((USBFS_frequencyChanged != 0) && (USBFS_transferState == USBFS_TRANS_STATE_IDLE))`
	`{`
	`/* Add core here.*/`
	`USBFS_frequencyChanged = 0;`
	`}`
	The USBFS_transferState variable is checked to make sure that the transfer completes.

USBFS_currentMute	Contains the mute configuration set by the host.
USBFS_currentVolume	Contains the volume level set by the host.

USBFS Audio Functional Description

Audio Class Request

This section describes the requests supported by the USBFS component. If a request is not supported, the USBFS component responds with a STALL, indicating a request error.

Class Request	USBFS Component Support Description	Device Class Definition for Audio - Section
SET_CUR	Interface: MUTE_CONTROL VOLUME_CONTROL	5.2.1.1
	Endpoint: SAMPLING_FREQ_CONTROL	
GET_CUR	Interface: MUTE_CONTROL VOLUME_CONTROL	5.2.1.2
	Endpoint: SAMPLING_FREQ_CONTROL	
GET_MIN	Interface: VOLUME_CONTROL	5.2.1.2
GET_MAX	Interface: VOLUME_CONTROL	5.2.1.2
GET_RES	Interface: VOLUME_CONTROL	5.2.1.2
GET_STAT	The content of the status message is reserved for future use. For now, a null packet should be returned in the data stage of the control transfer and the received null packet should be ACKed.	5.2.4.2

USBFS MIDI

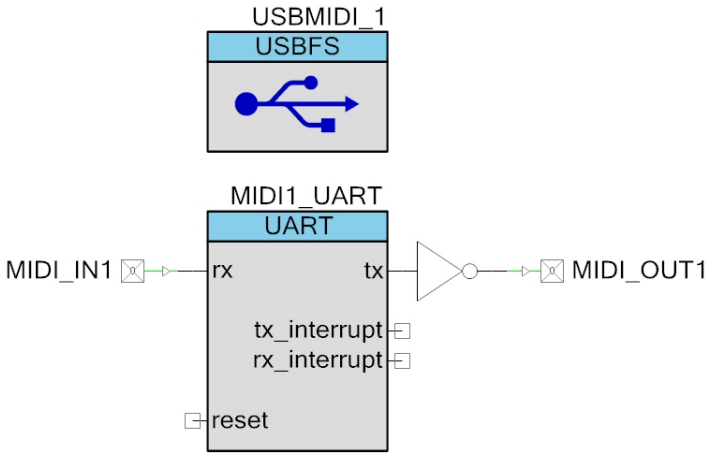
USBFS MIDI provides support for communicating with external MIDI equipment. It also provides support for the USB device class definition for MIDI devices. You can use this component to add MIDI I/O capability to a standalone device, or to implement MIDI capability for a host computer or mobile device through that computer or mobile device's USB port. In such cases, it presents itself to the host computer or mobile device as a class-compliant USB MIDI device and uses the native MIDI drivers in the host.

Features

- Provides USB MIDI Class Compliant MIDI input and output
- Supports hardware interfacing to external MIDI equipment using UART
- Provides adjustable transmit and receive buffers managed using interrupts
- Handles MIDI running status for both receive and transmit functions
- Supports up to 16 input and output ports using only two USB endpoints by using virtual cables.

The PSoC Creator Component Catalog contains a Schematic Macro implementation of a MIDI interface. The macro consists of instances of the UART component with the hardware MIDI interface configuration (31.25 kbps, 8 data bits) and a USBFS component with the descriptors configured to support MIDI devices. This allows the end user to use a MIDI-enabled USBFS component with minimal configuration changes.

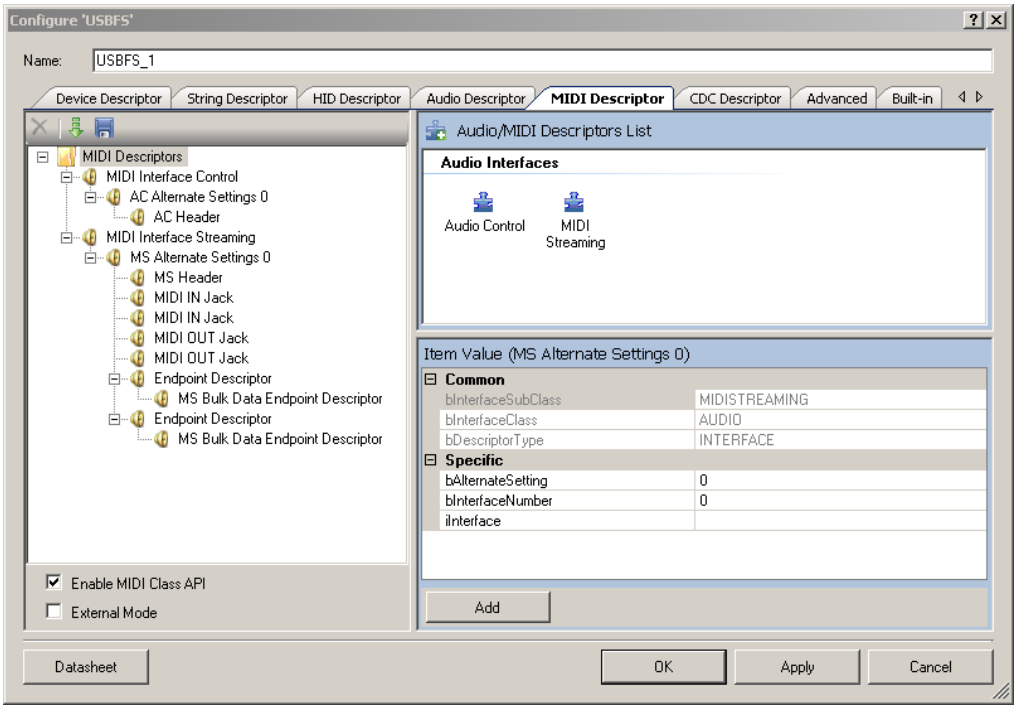
To start a MIDI-based project, drag the USBMIDI Schematic Macro labeled ‘USBMIDI’ from the Component Catalog onto your design. This macro has already been configured to function as an external mode MIDI device with 1 input and 1 output. See the Component Parameters section of this datasheet for information about modifying the parameters of this interface, such as the VID, PID, and String Descriptors.



The UART component is connected to digital input and output Pins components. The output pin is connected through the NOT gate to prepare the inverted signal to be supplied to the external transistor. Refer to the *MIDI 1.0 Detailed Specification* for more details about the hardware MIDI interface.

USBFS MIDI Parameters

To add and configure MIDI Streaming interface descriptors, open the Configure USBFS dialog and click the **MIDI Descriptor** tab.



To update the USBMIDI Schematic Macro for the external mode with 2 inputs and 2 outputs:

1. Go to the **MIDI Descriptor** tab of the USBMIDI_1 component.
2. Click the **Import MIDI Interface** button, browse to the following directory, and open the *USBMIDI 2x2.midi.xml* file.

   ```
   <INSTALL>\psoc\content\cycomponentlibrary\CyComponentLibrary.cylib\USBFS_v2.80\Custom\template\
   ```

3. Drag the UART Schematic Macro from the Component Catalog onto your design.
4. Configure the UART with the following options:

   Name: MIDI2_UART

   Mode: Full UART

Bits per second:	31250
Data bits:	8
Parity:	None
RX Buffer Size (bytes)	255
TX Buffer Size (bytes)	255

5. Connect the output pin through the NOT gate.

To Add MIDI Descriptors

1. Select the **MIDI Descriptors** root item in the tree on the left.
2. Under **Audio / MIDI Descriptors List** on the right, select either the **Audio Control** or **MIDI Streaming** interface.
3. Under **Item Value**, enter **bAlternateSetting** and **bInterfaceNumber** values as appropriate. Other fields are optional.

   Note These values are set manually. By contrast, for the general interface descriptors, these values are set automatically.
4. Click **Add** to add the descriptor to the tree on the left.

   You can rename the **MIDI Interface x** title by selecting a node and then clicking on it.

To Add Class-Specific Audio Control or MIDI Streaming Interface Descriptors

1. Select the appropriate **AC Alternate Settings x** or **MS Alternate Settings x** item in the tree on the left.
2. Under the **Audio / MIDI Descriptors List** on the right, select one of the items under **Audio Control Descriptors (1.0)**, **Audio Control Descriptors (2.0)**, or **MIDI Streaming Descriptors** as appropriate.

   Versions 1.0 and 2.0 refer to the versions of the corresponding specification document *Universal Serial Bus Device Class Definition for Audio Devices*.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add MIDI Endpoint Descriptors

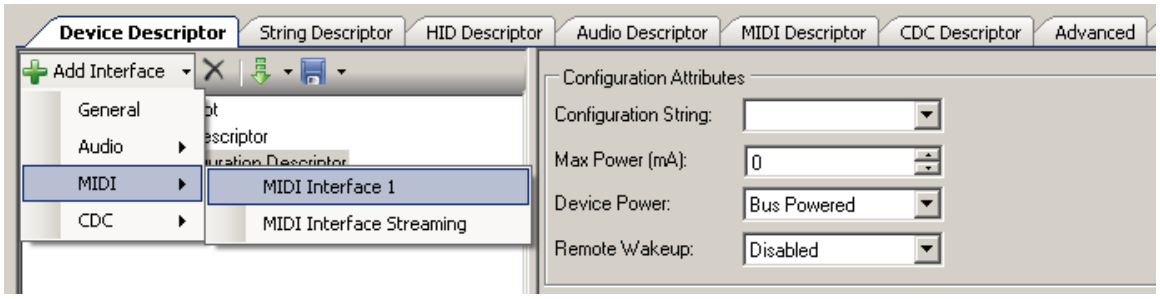
1. Select the appropriate **AC Alternate Settings x** or **MS Alternate Settings x** item in the tree on the left.
2. Under the **Audio / MIDI Descriptors List** on the right, select the **Endpoint Descriptor** item.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add Standard MS Bulk Data Endpoint Descriptor

1. Select the appropriate **Endpoint Descriptor** in the tree on the left.
2. Under the **Audio / MIDI Descriptors List** on the right, select **MS Endpoint Descriptor**.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add the Configured MIDI Interface Descriptor to the Device Descriptor Tree

1. Go to the **Device Descriptor** tab.
2. Select the **Configuration Descriptor** to which a new interface will belong.
3. Click the **Add Interface** tool button, choose **MIDI**, and select the appropriate item to add.



MIDI interfaces are disabled in the **Device Descriptor** tab list because they can only be edited on the **MIDI Descriptor** tab.

Note Click **Apply** or **OK** to save the changes on the various tabs. If you click **Cancel**, the descriptors you added will not be saved.

USBFS MIDI API Support

The following high-level APIs are available when the **Enable MIDI Class API** option in the **MIDI Descriptor** tab is selected.

Function	Description
USBMIDI_MIDI_EP_Init()	Initializes the MIDI interface and UARTs to be ready to receive data from the PC and MIDI ports.
USBMIDI_MIDI_IN_Service()	Services the USB MIDI IN endpoint.
USBMIDI_MIDI_OUT_EP_Service()	Services the USB MIDI OUT endpoint.
USBMIDI_PutUsbMidiIn()	Puts one MIDI message into the USB MIDI IN endpoint buffer. This is a MIDI input message to the host.
USBMIDI_callbackLocalMidiEvent()	Is a callback function from *USBMIDI_midi.c* to local processing in *main.c*.

Global Variables

Variable	Description
USBMIDI_midiInBuffer	Input endpoint buffer with a length equal to MIDI IN EP **Max Packet Size**. This buffer is used to save and combine the data received from the UARTs, generated internally by USBMIDI_PutUsbMidiIn() function messages, or both. The USBMIDI_MIDI_IN_Service() function transfers the data from this buffer to the PC.
USBMIDI_midiOutBuffer	Output endpoint buffer with a length equal to MIDI OUT EP **Max Packet Size**. This buffer is used by the USBMIDI_MIDI_OUT_EP_Service() function to save the data received from the PC. The received data is then parsed. The parsed data is transferred to the UARTs buffer and also used for internal processing by the USBMIDI_callbackLocalMidiEvent() function.
USBMIDI_midiInPointer	Input endpoint buffer pointer. This pointer is used as an index for the USBMIDI_midiInBuffer to write data. It is cleared to zero by the USBMIDI_MIDI_EP_Init() function.
USBMIDI_midi_in_ep	Contains the midi IN endpoint number, It is initialized after a SET_CONFIGURATION request based on a user descriptor. It is used in MIDI APIs to send data to the PC.
USBMIDI_midi_out_ep	Contains the midi OUT endpoint number. It is initialized after a SET_CONFIGURATION request based on a user descriptor. It is used in MIDI APIs to receive data from the PC.
USBMIDI_MIDI1_InqFlags USBMIDI_MIDI2_InqFlags	These optional variables are allocated when **External Mode** is enabled. The following flags help to detect and generate responses for SysEx messages.

Flag	Description
USBMIDI_INQ_SYSEX_FLAG	Non-real-time SysEx message received.
USBMIDI_INQ_IDENTITY_REQ_FLAG	Identity Request received. You should clear this bit when an Identity Reply message is generated.

void USBMIDI_MIDI_EP_Init(void)

Description: This function initializes the MIDI interface and UARTs to be ready to receive data from the PC and MIDI ports.

Parameters: None

Return Value: None

Side Effects: Changes the priority of the UARTs' TX and RX interrupts. For more information, see the Interrupt Priority section

void USBMIDI_MIDI_IN_Service(void)

Description: This function services the traffic from MIDI input ports (RX UARTs) or generated by the USBMIDI_PutUsbMidiIn() function and sends the data to the USBMIDI IN endpoint. It is non-blocking and should be called from the main foreground task. For more information about the usage of this API, see the USBFS MIDI Functional Description section.

This function is not protected from reentrant calls. When you must use this function in UART RX ISR to guaranty low latency, take care to protect it from reentrant calls.

Parameters: None

Return Value: None

Side Effects: None

void USBMIDI_MIDI_OUT_EP_Service(void)

Description: This function services the traffic from the USBMIDI OUT endpoint and sends the data to the MIDI output ports (TX UARTs). It is blocked by the UART when not enough space is available in the UART TX buffer.

This function is automatically called from OUT EP ISR in DMA with Automatic Memory Management mode. In Manual and DMA with Manual EP Management modes you must call it from the main foreground task.

Parameters: None

Return Value: None

Side Effects: None

uint8 USBMIDI_PutUsbMidiIn(uint8 ic, const uint8 midiMsg[], uint8 cable)

Description: This function puts one MIDI message into the USB MIDI In endpoint buffer. This is a MIDI input message to the host. This function is used only if the device has internal MIDI input functionality. The USBMIDI_MIDI_IN_Service() function should also be called to send the message from local buffer to the IN endpoint.

Parameters: uint8 ic: The length of the MIDI message or command is described on the following table.

Value	Description
0	No message (should never happen)
1-3	Complete MIDI message in midiMsg
3 - IN EP **Max Packet Size**	Complete SysEx message (without the EOSEX byte) in midiMsg
USBMIDI_MIDI_SYSEX	Start or continuation of SysEx message. Put event bytes in the midiMsg buffer
USBMIDI_MIDI_EOSEX	End of SysEx message. Put event bytes in the midiMsg buffer
USBMIDI_MIDI_TUNEREQ	Tune Request message (single-byte system common message)
0xF8 to 0xFF	Single-byte real-time message

const uint8 midiMsg[]: Pointer to MIDI message

uint8 cable: Cable number

Return Value:

Return Value	Description
USBMIDI_TRUE	Host is not ready to receive this message
USBMIDI_FALSE	Success transfer

Side Effects: None

void USBMIDI_callbackLocalMidiEvent(uint8 cable, uint8 midiMsg)*

Description: This is a callback function that locally processes data received from the PC in *main.c*. You should implement this function if you want to use it. It is called from the USB output processing routine for each MIDI output event processed (decoded) from the output endpoint buffer.

Parameters: uint8 cable: Cable number

uint8* midiMsg: Pointer to the 3-byte MIDI message

Return Value: None

Side Effects: None

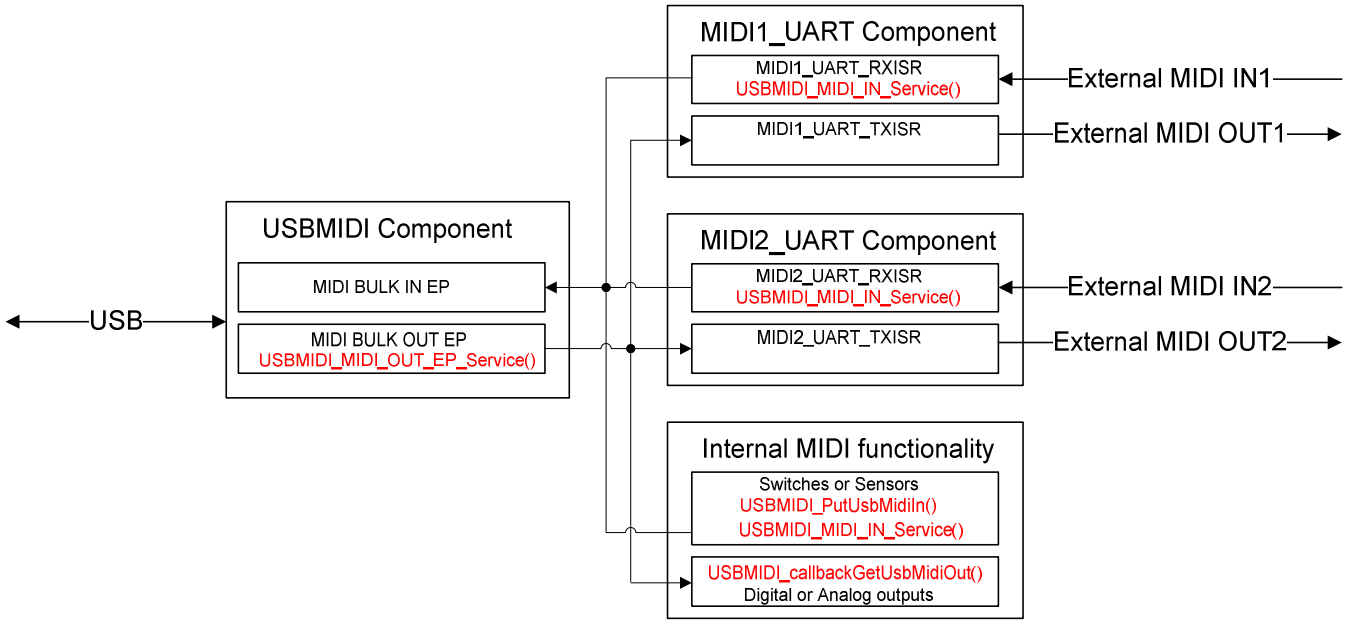
USBFS MIDI Functional Description

The MIDI descriptor tab allows you to easily create a MIDI interface device with one or more sets of physical MIDI ports (you may have to place and configure instances of a UART component). It handles all details of sending and receiving MIDI messages to external MIDI equipment. This is referred to as external MIDI functionality and is an optional setting in the component.

The MIDI implementation internally handles running status when communicating with external MIDI equipment. Running status is automatically implemented on the output to reduce serial data traffic, and running status in managed on the input to correctly assemble complete MIDI messages when the external MIDI equipment is sent using running status. Refer to *MIDI 1.0 Detailed Specification* for more details about Running Status feature.

Figure 3 shows the external mode USB-MIDI interface with two inputs and two outputs.

Figure 3. External Mode USB-MIDI Interface



Implementing external functionality requires you to place and configure UART components with the names “MIDI1_UART” and “MIDI2_UART”. These hardcoded names allow the USBMIDI component to call UART APIs and automatically transfer received data from the host messages to the external MIDI port. In Manual and DMA with Manual EP management mode, you must call the USBMIDI_MIDI_OUT_EP_Service() API from the main loop.

For the opposite direction, to service MIDI event data from the UART components you must call the USBMIDI_MIDI_IN_Service() API in the main loop for Manual and DMA with Manual memory management mode. For DMA with Automatic mode, call this function from the user section(MIDI[1..2]_UART_RXISR_END) of the Interrupt Service Routine for the RX portion of the UART(MIDI[1..2]_UART_RXISR).

You can use local switches and sensors to create MIDI messages for the host (use the USBMIDI_PutUsbMidiIn() function). MIDI messages from the host can directly control local functions such as digital and analog outputs (implement the USBMIDI_callbackLocalMidiEvent() function, which is called to process all received messages).

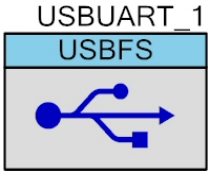
Interrupt Priority

The data received from the host is serviced inside the MIDI BULK OUT EP ISR. When you select a small UART **TX Buffer Size**, the code waits for the UART transmit operation to complete and continues filling the TX buffer. The priority of the Interrupt Service Routine for the TX portion of the UART should be higher than the MIDI BULK OUT EP ISR priority. The USBMIDI_MIDI_EP_Init() function automatically changes the default priority for the mentioned interrupt to the USBMIDI_CUSTOM_UART_TX_PRIOR_NUM value. Cypress recommends that you select UART **TX Buffer Size** to be the same or greater than MIDI BULK OUT EP **Max Packet Size**. The optimal **Max Packet Size** is 32.

The priority of the UART RX ISR should be higher than TX ISR so that the four bytes of hardware FIFO overloads are not allowed. The optimal UART **RX Buffer Size** is 255. The USBMIDI_MIDI_EP_Init() function automatically changes the default priority for the UART RX interrupt to the USBMIDI_CUSTOM_UART_RX_PRIOR_NUM value. The USBMIDI_MIDI_EP_Init() function automatically changes the default priority for the UART RX interrupt to the USBMIDI_CUSTOM_UART_RX_PRIOR_NUM value.

USBUART (CDC)

The PSoC Creator Component Catalog contains a Schematic Macro implementation of a communications device class (CDC) interface (also known as USBUART). This is a USBFS component with the descriptors configured to implement a CDC interface. This allows you to use a CDC-enabled USBFS component with minimal configuration required.



To start a USBUART-based project, drag the USBUART Schematic Macro labeled ‘USBUART (CDC Interface)’ from the Component Catalog onto your design. This macro has already been configured to function as a CDC device. See the Component Parameters section of this datasheet for information about modifying the parameters of this interface, such as the VID, PID, and String Descriptors.

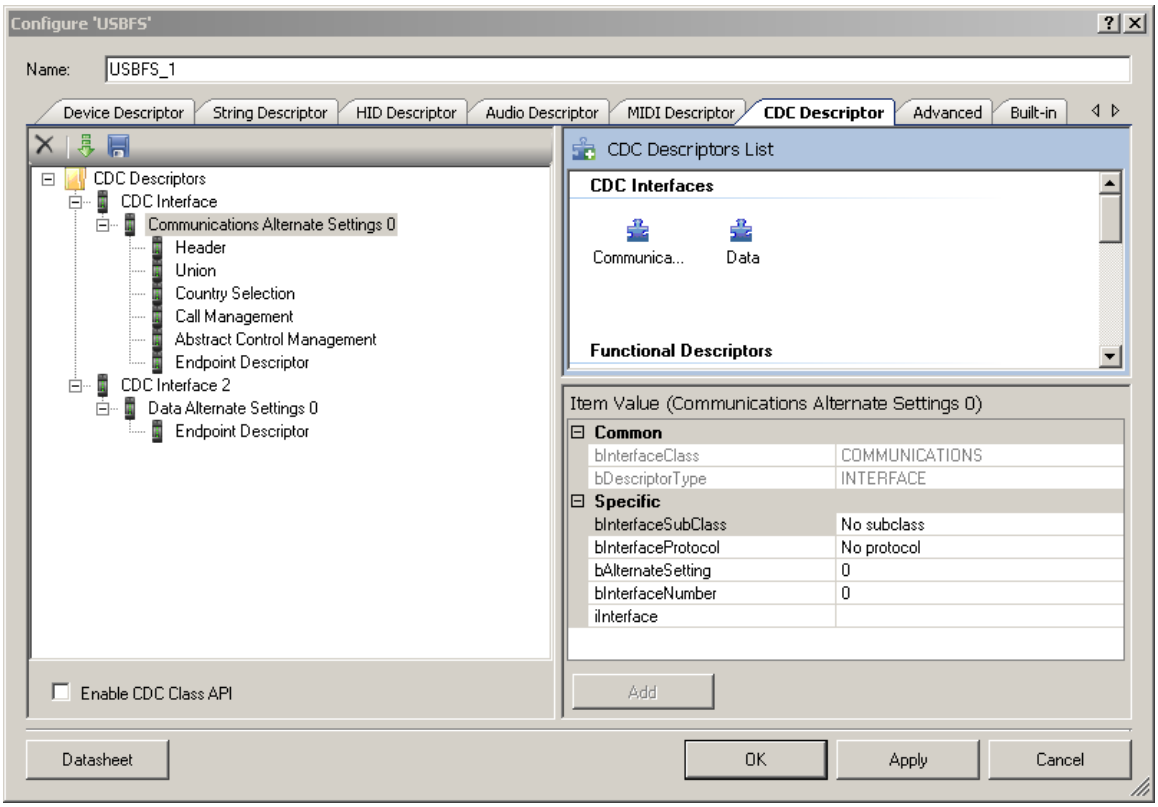
The CDC device requires drivers to be installed. The drivers can be found in the generated sources folder of your project:

<PROJECT_NAME>.cydsn\Generated_Source\PSoC5\<INSTANCE_NAME>_cdc.inf

See the USB_UART example project for detailed steps on how to install a driver.

USBUART Parameters

To add and configure communications and data interface descriptors for the USBUART, open the Configure USBFS dialog and click the **CDC Descriptor** tab.



To Add CDC Descriptors

1. Select the **CDC Descriptors** root item in the tree on the left.
2. Under the **CDC Descriptors List** on the right, select either the **Communications** or **Data** interface.
3. Under **Item Value**, enter **bAlternateSetting** and **bInterfaceNumber** values as appropriate. Other fields are optional.

   Note These values are set manually. By contrast, for the general interface descriptors these values are set automatically.
4. Click **Add** to add the descriptor to the tree on the left.
5. You can rename the **CDC Interface x** title by selecting a node and clicking on it.

To Add Functional Descriptors

1. Select the appropriate **Communications Alternate Settings x** item in the tree on the left.
2. Under the **CDC Descriptors List** on the right, select one of the items under **Functional Descriptors** as appropriate.

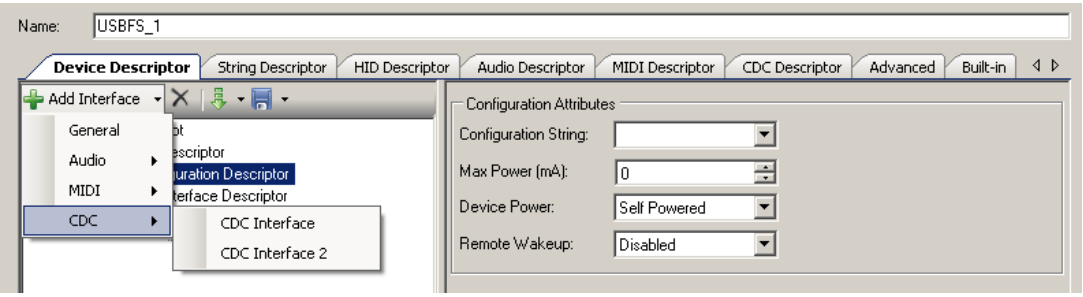
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add Endpoint Descriptors

1. Select the appropriate **Communications Alternate Settings x** or **Data Alternate Settings x** item in the tree on the left.
2. Under the **CDC Descriptors List** on the right, select the **Endpoint Descriptor** item.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add** to add the descriptor to the tree on the left.

To Add the Configured CDC Interface Descriptor to the Device Descriptor Tree

1. Go to the **Device Descriptor** tab.
2. Select the **Configuration Descriptor** to which a new interface will belong.
3. Click the **Add Interface** tool button, choose **CDC**, and select the appropriate item to add.



CDC interfaces are disabled in the **Device Descriptor** tab list because they can only be edited on the **CDC Descriptor** tab.

Note Click **Apply** or **OK** to save the changes on the various tabs. If you click **Cancel**, the descriptors you added will not be saved.

USBUART (CDC) API Support

The following high-level APIs are available when the **Enable CDC Class API** option in the **CDC Descriptor** tab is selected. These APIs do not support DMA with Automatic Memory Management.

Function	Description
USBUART_CDC_Init()	Initializes the CDC interface to be ready for the receive data from the PC
USBUART_PutData()	Sends a specified number of bytes from the location specified by a pointer to the PC
USBUART_PutString()	Sends a null terminated string to the PC
USBUART_PutChar()	Writes a single character to the PC

Function	Description
USBUART_PutCRLF()	Sends a carriage return (0x0D) and line feed (0x0A) to the PC
USBUART_GetCount()	Returns the number of bytes that were received from the PC
USBUART_CDCIsReady()	Returns a nonzero value if the component is ready to send more data to the PC
USBUART_DataIsReady()	Returns a nonzero value if the component received data or received a zero-length packet
USBUART_GetData()	Gets a specified number of bytes from the input buffer and places them in a data array specified by the passed pointer
USBUART_GetAll()	Gets all bytes of received data from the input buffer and places them into a specified data array
USBUART_GetChar()	Reads one byte of received data from the buffer
USBUART_IsLineChanged()	Returns the clear-on-read status of the line
USBUART_GetDTERate()	Returns the data terminal rate set for this port in bits per second
USBUART_GetCharFormat()	Returns the number of stop bits
USBUART_GetParityType()	Returns the parity type for the CDC port
USBUART_GetDataBits()	Returns the number of data bits for the CDC port
USBUART_GetLineControl()	Returns the line control bitmap

Global Variables

Variable	Description
USBUART_lineCoding	Contains the current line coding structure. The host sets it using a SET_LINE_CODING request and returns it to the user code using the USBUART_GetDTERate(), USBUART_GetCharFormat(), USBUART_GetParityType(), and USBUART_GetDataBits() APIs.
USBUART_lineControlBitmap	Contains the current control-signal bitmap. The host sets it using a SET_CONTROL_LINE request and returns it to the user code using the USBUART_GetLineControl() API.
USBUART_lineChanged	Used as a flag for the USBUART_IsLineChanged() API, to inform it that the host has been sent a request to change line coding or control bitmap.
USBUART_cdc_data_in_ep	Contains the data IN endpoint number. It is initialized after a SET_CONFIGURATION request based on a user descriptor. It is used in CDC APIs to send data to the PC.
USBUART_cdc_data_out_ep	Contains the data OUT endpoint number. It is initialized after a SET_CONFIGURATION request based on user descriptor. It is used in CDC APIs to receive data from the PC.

void USBUART_CDC_Init(void)

Description: This function initializes the CDC interface to be ready to receive data from the PC. This API should be called after the device has been started and configured using USBUART_Start() API to initialize and start the USBFS component operation. Then call the USBUART_GetConfiguration() API to wait until the host has enumerated and configured the device. For example:

```
USBUART_Start(…);
while(USBUART_GetConfiguration() == 0){};
USBUART_CDC_Init();
```

Parameters: None

Return Value: None

Side Effects: None

void USBUART_PutData(const uint8* pData, uint16 length)

Description: This function sends a specified number of bytes from the location specified by a pointer to the PC. The USBUART_CDCIsReady() function should be called before sending new data, to be sure that the previous data has finished sending.

If the last sent packet is less than maximum packet size the USB transfer of this short packet will identify the end of the segment. If the last sent packet is exactly maximum packet size, it shall be followed by a zero-length packet (which is a short packet) to assure the end of segment is properly identified. To send zero-length packet, use USBUART_PutData() API with length parameter set to zero.

Parameters: const uint8* pData: Pointer to the buffer containing data to be sent

uint16 length: Specifies the number of bytes to send from the pData buffer. Maximum length is limited to 64 bytes. Data will be lost if length is greater than Max Packet Size.

Return Value: None

Side Effects: None

void USBUART_PutString(const char8* string[])

Description: This function sends a null terminated string to the PC. This function will block if there is not enough memory to place the whole string. It will block until the entire string has been written to the transmit buffer. The USBUART_CDCIsReady() function should be called before sending data with a new call to USBUART_PutString(), to be sure that the previous data has finished sending. This function sends zero-length packet automatically, if the length of the last packet, sent by this API, is equal to **Max Packet Size**.

Parameters: const char8 string[]: Pointer to the string to be sent to the PC.

Return Value: None

Side Effects: None

void USBUART_PutChar(char8 txDataByte)

Description: This function writes a single character to the PC at a time. This is an inefficient way to send large amounts of data.

Parameters: char8 txDataByte: Character to be sent to the PC

Return Value: None

Side Effects: None

void USBUART_PutCRLF(void)

Description: This function sends a carriage return (0x0D) and line feed (0x0A) to the PC. This API is provided to mimic API provided by our other UART components.

Parameters: None

Return Value: None

Side Effects: None

uint16 USBUART_GetCount(void)

Description: This function returns the number of bytes that were received from the PC. The returned length value should be passed to USBUART_GetData() as a parameter to read all received data. If all of the received data is not read at one time by the USBUART_GetData() API, the unread data will be lost.

Parameters: None

Return Value: uint16: Returns the number of received bytes. The maximum amount of received data at a time is limited to 64 bytes.

Side Effects: None

uint8 USBUART_DataIsReady(void)

Description: This function returns a nonzero value if the component received data or received a zero-length packet. The USBUART_GetAll() or USBUART_GetData() API should be called to read data from the buffer and reinitialize the OUT endpoint even when a zero-length packet is received. These APIs will return zero value when zero-length packet is received.

Parameters: None

Return Value: uint8: If the OUT packet is received, this function returns a nonzero value. Otherwise, it returns zero.

Side Effects: None

uint8 USBUART_CDCIsReady(void)

Description: This function returns a nonzero value if the component is ready to send more data to the PC; otherwise, it returns zero. The function should be called before sending new data when using any of the following APIs: USBUART_PutData(), USBUART_PutString(), USBUART_PutChar or USBUART_PutCRLF(), to be sure that the previous data has finished sending.

Parameters: None

Return Value: uint8: If the buffer can accept new data, this function returns a nonzero value. Otherwise, it returns zero.

Side Effects: None

uint16 USBUART_GetData(uint8* pData, uint16 length)

Description: This function gets a specified number of bytes from the input buffer and places them in a data array specified by the passed pointer. The USBUART_DataIsReady() API should be called first, to be sure that data is received from the host. If all received data will not be read at once, the unread data will be lost. The USBUART_GetData() API should be called to get the number of bytes that were received.

Parameters: uint8* pData: Pointer to the data array where data will be placed

uint16 length: Number of bytes to read into the data array from the RX buffer. The maximum length is limited by the number of received bytes or 64 bytes.

Return Value: uint16: Number of bytes which function moves from endpoint RAM into the data array. The function moves fewer than the requested number of bytes if the host sends fewer bytes than requested or sends zero-length packet.

Side Effects: None

uint16 USBUART_GetAll(uint8* pData)

Description: This function gets all bytes of received data from the input buffer and places them into a specified data array. The USBUART_DataIsReady() API should be called first, to be sure that data is received from the host.

Parameters: uint8* pData: Pointer to the data array where data will be placed.

Return Value: uint16: Number of bytes received. The maximum amount of the received at a time data is 64 bytes.

Side Effects: None

uint8 USBUART_GetChar(void)

Description: This function reads one byte of received data from the buffer. If more than one byte has been received from the host, the rest of the data will be lost.

Parameters: None

Return Value: uint8: Received one character

Side Effects: None

uint8 USBUART_IsLineChanged(void)

Description: This function returns the clear-on-read status of the line. It returns not zero value when the host sends updated coding or control information to the device. The USBUART_GetDTERate(), USBUART_GetCharFormat() or USBUART_GetParityType() or USBUART_GetDataBits() API should be called to read data coding information. The USBUART_GetLineControl() API should be called to read line control information.

Parameters: None

Return Value: uint8: If SET_LINE_CODING or CDC_SET_CONTROL_LINE_STATE requests are received, it returns a nonzero value. Otherwise, it returns zero.

Return Value	Description
USBUART_LINE_CODING_CHANGED	Line coding changed
USBUART_LINE_CONTROL_CHANGED	Line control changed

Side Effects: None

uint32 USBUART_GetDTERate(void)

Description: This function returns the data terminal rate set for this port in bits per second.

Parameters: None

Return Value: uint32: Returns a value of the data rate in bits per second

Side Effects: None

uint8 USBUART_GetCharFormat(void)

Description: This function returns the number of stop bits.

Parameters: None

Return Value: uint8: Returns the number of stop bits.

Return Value	Description
USBUART_1_STOPBIT	1 stop bit
USBUART_1_5_STOPBITS	1,5 stop bits
USBUART_2_STOPBITS	2 stop bits

Side Effects: None

uint8 USBUART_GetParityType(void)

Description: This function returns the parity type for the CDC port.

Parameters: None

Return Value: uint8:

Return Value	Description
USBUART_PARITY_NONE	None
USBUART_PARITY_ODD	Odd
USBUART_PARITY_EVEN	Even
USBUART_PARITY_MARK	Mark
USBUART_PARITY_SPACE	Space

Side Effects: None

uint8 USBUART_GetDataBits(void)

Description: This function returns the number of data bits for the CDC port.

Parameters: None

Return Value: uint8: Returns the number of data bits. The number of data bits can be 5, 6, 7, 8, or 16.

Side Effects: None

uint16 USBUART_GetLineControl(void)

Description: This function returns the line control bitmap that the host sends to the device.

Parameters: None.

Return Value: uint8:

Return Value	Notes
USBUART_LINE_CONTROL_DTR	Indicates that a DTR signal is present. This signal corresponds to V.24 signal 108/2 and RS232 signal DTR.
USBUART_LINE_CONTROL_RTS	Carrier control for half-duplex modems. This signal corresponds to V.24 signal 105 and RS232 signal RTS.
RESERVED	The rest of the bits are reserved.

Note Some terminal emulation programs do not properly handle these control signals. They update information about DTR and RTS state only when the RTS signal changes the state.

Side Effects: None

USBUART Functional Description

CDC Class Request

This section describes the requests supported by the USBUART component. If a request is not supported, the USBUART component responds with a STALL, indicating a request error.

Class Request	USBUART Component Support Description	Communications Class Subclass Specification for PSTN Devices
SET_LINE_CODING	Allows the host to specify typical asynchronous line-character formatting properties such as: data terminal rate, number of stop bits, parity type and number of data bits. It applies to data transfers both from the host to the device and from the device to the host.	6.3.10
GET_LINE_CODING	Allows the host to find out the currently configured line coding.	6.3.11
SET_CONTROL_LINE_STATE	Generates RS-232/V.24 style control signals – RTS and DTR.	6.3.12

Not supported by the USBUART component request related to USBUART:

Class Request	Description	Communications Class Subclass Specification for PSTN Devices
SERIAL_STATE	Allows the host to read the current state of the carrier detect (CD), DSR, break, and ring signal (RI).	6.5.4

Code Example (CE60246) USBUART Migration

Before the addition of USBUART CDC support in the USBFS v2.0 component (available in PSoC Creator 2.0 or later), a USBUART component was available as a Code Example component in *CE60246 - USBUART in PSoC® 3 / PSoC 5*. This Code Example USBUART is no longer supported and you are encouraged to migrate to the official component. This section details the steps required to complete this migration.

Schematic

1. Open your existing design in PSoC Creator 2.0 or later.
2. Take note of your existing component name, Vendor ID, Product ID, Device Release, Manufacturer String, and Product String in your existing USBUART component.
3. Delete your existing USBUART component.
4. Place a 'USBUART (CDC Interface)' component from the PSoC Creator Component Catalog onto your design.
5. Open the new component and configure the component with the parameters noted from the previous USBUART design. See the Component Parameters section of this datasheet for details about how to enter the VID, PID, and various device strings into the new component.

API

Table 1 outlines the required API changes to migrate from the CE60246 USBUART to the USBFS v2.0+ version of the USBUART. Most changes are minor modifications and should have a minimal effect on the existing project. Note that the USBFS v2.0+ version of the USBUART includes a larger selection of CDC-specific APIs (see the CDC Class Support API list earlier in the datasheet).

Table 1. API Migration

CE60246 API	USBFS v2.0+ API	Changes Required in Migration
void USBUART_1_Init (void)	void USBUART_1_CDC_Init (void)	▪ API name change
uint8 USBUART_1_bGetRxCount (void)	uint16 USBUART_1_GetCount (void)	▪ API name change ▪ Return value changed from uint8 to uint16
void USBUART_1_ReadAll (uint8* pData)	uint16 USBUART_1_GetAll (uint8* pData)	▪ API name change ▪ Return value changed from void to uint16
void USBUART_1_Write (uint8 *pData, uint8 bLength)	void USBUART_1_PutData (const uint8* pData, uint16 length)	▪ API name change ▪ Length parameter type changed from uint8 to uint16
uint8 USBUART_1_bTxIsReady (void)	uint8 USBUART_1_CDCIsReady (void)	▪ API name change

Note The table assumes the component name is “USBUART_1”

Resources

USB is implemented as a fixed-function block. The component utilizes 6 Interrupts and 2 Pins.

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ TA ≤ 85 °C and TJ ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

USB DC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
V_{USB_5}	Device supply for USB operation	USB configured, USB regulator enabled	4.35	–	5.25	V
$V_{USB_3.3}$		USB configured, USB regulator bypassed	3.15	–	3.6	V
V_{USB_3}		USB configured, USB regulator bypassed	2.85	–	3.6	V
$I_{USB_Configured}$	Device supply current in device active mode, bus clock and IMO = 24 MHz	V_{DDD} = 5 V	–	10	–	mA
		V_{DDD} = 3.3 V	–	8	–	mA

Parameter	Description	Conditions	Min	Typ	Max	Units
$I_{USB_Suspended}$	Device supply current in device sleep mode	V_{DDD} = 5 V, connected to USB host, PICU configured to wake on USB resume signal	–	0.5	–	mA
		V_{DDD} = 5 V, disconnected from USB host	–	0.3	–	mA
		V_{DDD} = 3.3 V, connected to USB host, PICU configured to wake on USB resume signal	–	0.5	–	mA
		V_{DDD} = 3.3 V, disconnected from USB host	–	0.3	–	mA

USB Driver AC Specifications

Parameter	Description	Conditions	Min	Typ	Max	Units
Tr	Transition rise time		–	–	20	ns
Tf	Transition fall time		–	–	20	ns
TR	Rise/fall time matching		90%	–	111%	
V_{CRS}	Output signal crossover voltage		1.3	–	2	V

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
2.80	In the USB HID configuration, the “RPT_TABLE” arrays were modified: empty entries {NULL, NULL, NULL} replaced with {0x00u, NULL, NULL}.	The ARM GCC 4.8.4 compiler generated a warning when using HID Report Descriptors with the REPORT_ID item.
	Edited the datasheet.	Rearranged sections to conform to the template.
2.70	Fixed respond to GET_DESCRIPTOR request. If a device does not support a requested descriptor, it responds with a Request Error.	USB Command Verifier version 1.4.10.2 fails the chapter 9 and HID tests.
	Fixed rare IN endpoint transaction fault in DMA w/Automatic Memory Management mode.	The fix consumes additional hardware resources. The epDMAautoOptimization parameter in the expression view of the Device Descriptor tab enables resource optimization. Set parameter value to true only when a single IN endpoint is present in the device.

Version	Description of Changes	Reason for Changes / Impact
	USBUART_lineCoding array initialized with following default configuration: 115200 baud, 8 data bits, None parity, 1 stop bit.	The first GET_LINE_CODING request does not send proper configuration to the terminal software.
	Modified USBUART_PutString() API to send zero-length packet automatically, if the length of the last packet, sent by this API, is equal to maximum packet size.	If the last sent packet is exactly maximum packet size, it shall be followed by a zero-length packet.
	Updated USBFS_LoadInEP() API description.	Clarified the process to set data ready status.
2.60	Interface Association Descriptor support has been added.	The Interface Association Descriptor has been implemented as described in *USB ECN: Interface Association Descriptors* documentation.
	Quick import of HID templates feature added.	Usability improvements.
	Added possibility to import HID report descriptors that were created using the official USB-IF HID Descriptor Tool.	
	Updated MISRA Compliance section.	The component verified for MISRA-C:2004 coding guidelines compliance and has component specific deviations.
	Added optional vbusdet input.	This input provides the ability to connect VBUS for power monitoring.
	Fixed inaccessibility of Interface Protocol field of Interface descriptor and also Synch Type and Usage Type fields of Endpoint descriptor in some cases.	
2.50	Editing of HID report's name and comments in the customizer is available via context menu in the tree, "Rename" command.	Simplify editing of HID report descriptor.
	Fixed USBFS_SetEndpointHalt() and USBFS_ClearEndpointHalt() functions.	This fix allows continue data transfer when the host clears the ENDPOINT_HALT feature.
	Added MISRA Compliance section.	This component was not verified for MISRA-C:2004 coding guidelines compliance.
2.40	Fixed rare IN endpoint transaction fault and dynamic endpoint reconfiguration in DMA w/Automatic Memory Management mode.	
	The field that displays the device number was added to the device descriptor in the device descriptor tab.	This number has to be used in USBFS_Start() API as a parameter.
	Added DMA w/Manual Memory Management support for PSoC 5 silicon.	

Version	Description of Changes	Reason for Changes / Impact
	DRC with error is generated when Bulk endpoint MaxPacketSize value is not from the list {8, 16, 32, 64}	This error is also checked in customizer. User won't be able to close the customizer if the wrong value is entered.
	Added multiple Report ID support for HID descriptor.	
2.30	Fixed unexpected endpoint reconfiguration after SET_INTERFACE request sent to interface not related to affected endpoint.	A device with multiple interfaces and alternate settings must reconfigure only endpoints which are required by SetInterface request.
	Added user code sections(`#START`...`#END`) to the SET_CUR/GET_CUR Audio class requests handler.	To let the user update volume control requests handler with multi-channel audio volume control support.
2.20	Added PSoC 5LP silicon support.	
	Updated characterization data.	
	Minor datasheet edits.	
2.12	Added MIDI devices support: ▪ Added the new “MIDI Descriptor” tab. This tab allows the user to configure MIDI descriptors. ▪ Optional high level APIs.	The MIDI interface has been implemented as described in *Universal Serial Bus Device Class Definition for MIDI Devices v1.0* documentation.
	Added the USBFS_Resume_Condition() API for PSoC 5 only device to check the condition for resume.	A PSoC 5 device has neither PICU wakeup source nor standard D+ pin APIs to check the condition for waking up. This function reads the D+ pin level through USBIO block and returns the resume condition.
	Reorganized the datasheet.	
2.11	Added all USBFS APIs with the CYREENTRANT keyword when they are included in the .cyre file.	Not all APIs are truly reentrant. Comments in the component API source files indicate which functions are candidates. This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections.
	The data toggle is always set to DATA0 when performing an IN data transfer for an isochronous endpoint.	According to the USB 2.0 specification for Isochronous Transactions, a full-speed device should only send DATA0 PIDs in data packets.
	Fixed the Stop_DMA function to free all of the endpoint DMA TDs used for Mode 3 operation.	This function stopped only one channel.
	Changed default driver mode for the VBUS monitor input pin to High Impedance and removed the suppressing API generation for this pin.	This change allows you to reduce power consumption for low power projects.

Version	Description of Changes	Reason for Changes / Impact
2.10	Fixed handling of the class-specific requests in USBFS_DispatchClassRqst() function.	The Audio requests were stalled.
2.0	Added CDC class support: ▪ Added new “CDC Descriptor” tab. This tab allows the user to configure CDC descriptors. ▪ SET_LINE_CODING/GET_LINE_CODING CLR_CUR/SET_CONTROL_LINE_STATE CDC class request support. ▪ Optional high level APIs.	The CDC interface has been implemented as described in Section 4 of the *USB Class Definitions for Communications Devices v1.2* documentation.
	Added Audio Class 2.0 class support. On the “Audio” tab, added two new groups of available descriptors. They are called “Audio Control Descriptors (2.0)” and “Audio Streaming Descriptors (2.0)”. Existing groups “Audio Control Descriptors” and “Audio Streaming Descriptors” were renamed to “Audio Control Descriptors (1.0)” and “Audio Streaming Descriptors (1.0)”.	New descriptors represent *USB Device Class Definition for Audio Devices release 2.0* specification.
	Added DMA transfers implementation: ▪ Mode2: Manual DMA with Manual Memory Management ▪ Mode3: Auto DMA with Auto Memory Management ▪ USBFS_InitEP_DMA() API has been added. ▪ USBFS_LoadInEP()/USBFS_ReadOutEP() APIs modified to support DMA transfers.	DMA transaction releases the CPU use during data transfers.

Version	Description of Changes	Reason for Changes / Impact

	Added function USBFS_IsConfigurationChanged().	Win 7 OS could send double SET_CONFIGURATION requests with same configuration number. In this case user-level code should re-enable OUT endpoints after each request.
		This function should be used to detect that configuration has been changed from the PC. If it returns a nonzero value, the USBFS_GetConfiguration() API is can be used to get the configuration number.
		Usage model in main loop:
		`if(USBFS_IsConfigurationChanged() != 0)`
		`{`
		`  if(USBFS_GetConfiguration() != 0)`
		`  {`
		`  USBFS_EnableOutEP(OUT_EP);`
		`  }`
		`}`

	Fixed issue with Wakeup from Sleep mode.	USB_BUS_RST_CNT register is nonretention and should be reloaded after sleep mode for correct USB enumeration of PSoC 3 ES2 and PSoC 5 silicon.
	Moved the endpoint memory management group box from the device options panel to the root device options panel.	Endpoint memory management settings should be global for whole configuration. In the previous version these settings were individual for each device descriptor.
1.60	Added function USBFS_TerminateEP(uint8 ep) to NAK an endpoint.	This function can be used before endpoint reconfiguration or device mode switching.
	Initialized USBFS_hidProtocol variable to HID_PROTOCOL_REPORT value in USBFS_InitComponent() and USBFS_reInitComponent() functions.	To comply with HID “7.2.6 Set_Protocol Request” --- “When initialized, all devices default to report protocol.”
	Added support for SET_FEATURE/CLR_FEATURE requests to an interface.	For passing WHQL test.
	Added logic to the SET_IDLE request handling to support proper timing.	To comply with HID "7.2.4 Set_Idle Request"
	Added support for Audio class requests: SET_CUR/CLR_CUR to an interface and Endpoint for Sampling Frequency, Mute, and Volume controls.	To comply with Audio Class Definition “5.2.1.1 Set Request” and “5.2.1.2 Get Request”
	Renamed Bootloader APIs to have instance name first. Added the backward compatible defines.	Preparation for future ability to boot from multiple interfaces.
	Added characterization data to datasheet	

Version	Description of Changes	Reason for Changes / Impact
	Minor datasheet edits and updates	
1.50.a	Made datasheet change log cumulative	Customer convenience.
1.50	Added USB Suspend, Resume, and Remote Wakeup functionality.	The USB device should support suspend and resume functionality.
	Renamed most APIs to remove Hungarian notation, old names are supported for backward compatibility.	To comply with corporate coding standards.
	Added GET_INTERFACE/SET_INTERFACE requests support.	A device must support the GetInterface/SetInterface requests if it has alternate settings for that interface.
	Integrated specific APIs to support the bootloader: CyBtldrCommStart, CyBtldrCommStop, CyBtldrCommReset, CyBtldrCommWrite, CyBtldrCommRead.	USB could be used as a communication component for the Bootloader with this feature.
	Added generic USB Bulk Wraparound Transfer example to datasheet.	Described generic USB usage for user.
	Added the extern_cls and extern_vnd parameters to the Advanced tab of the Configure dialog.	These parameters enable other components at the solutions level, to provide their handling of Vendor and Class requests themselves.
	Restriction has been added to DMA w/Manual Memory Management section.	This restriction shows how to properly use Mode 2/3 transfers.
	Modified 'Advanced' tab layout.	Replaced the data grid with check boxes with information about each parameter to improve usability.
	Added Audio Descriptors tab to the Configure dialog.	This allows you to add and configure audio descriptors for your component.
	Removed SOF ISR enable/disable from Start/Stop APIs.	SOF interrupts occur each 1 ms, but were not used by the component. If an application requires this interrupt, it can be enabled by calling: `CyIntEnable(USBFS_SOF_VECT_NUM);`
1.30.b	Added information to the component that advertizes its compatibility with silicon revisions.	The tool reports an error/warning if the component is used on incompatible silicon. If this happens, update to a revision that supports your target device.
1.30.a	Moved local parameters to formal parameter list.	To address a defect that existed in PSoC Creator v1.0 Beta 4.1 and earlier, the component was updated so that it could continue to be used in newer versions of the tool. This component used local parameters, which are not exposed to the user, to do background calculations on user input. These parameters have been changed to formal parameters which are visible, but not editable. There are no functional changes to the component but the affected parameters are now visible in the “expression view” of the customizer dialog.

Version	Description of Changes	Reason for Changes / Impact
1.30	Updated the Configure dialog and datasheet.	Added the Enable SOF Output parameter to the Advanced tab of the Configure dialog. Updated the USBFS_ReadOutEP() function in the datasheet to reflect the correct return value.
1.20.b	Added information to the component that advertizes its compatibility with silicon revisions.	The tool reports an error/warning if the component is used on incompatible silicon. If this happens, update to a revision that supports your target device.
1.20.a	Moved local parameters to formal parameter list.	To address a defect that existed in PSoC Creator v1.0 Beta 4.1 and earlier, the component was updated so that it could continue to be used in newer versions of the tool. This component used local parameters, which are not exposed to the user, to do background calculations on user input. These parameters have been changed to formal parameters which are visible, but uneditable. There are no functional changes to the component but the affected parameters are now visible in the “expression view” of the customizer dialog.
1.10.b	Added information to the component that advertizes its compatibility with silicon revisions.	The tool reports an error/warning if the component is used on incompatible silicon. If this happens, update to a revision that supports your target device.
1.10.a	Moved local parameters to formal parameter list.	To address a defect that existed in PSoC Creator v1.0 Beta 4.1 and earlier, the component was updated so that it could continue to be used in newer versions of the tool. This component used local parameters, which are not exposed to the user, to do background calculations on user input. These parameters have been changed to formal parameters which are visible, but un-editable. There are no functional changes to the component but the affected parameters are now visible in the “expression view” of the customizer dialog.

© Cypress Semiconductor Corporation, 2014. 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® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks 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.