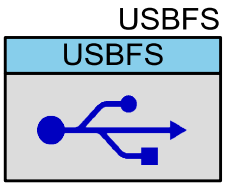
Full Speed USB (USBFS)

2.10

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
- Optional CDC class support

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) For the PSoC 3:
      i) **ILO**: Select 100 kHz.
      ii) **IMO**: Select Osc 24.000 MHz.
      iii) **USB**: Enable and select IMOx2 – 48.000 MHz.
   b) For the PSoC 5:
      i) **ILO**: Select 100 kHz.
      ii) XTAL: Enable and configure Freq: 24 MHz. Make sure the external 24 MHz crystal is installed on the DVK.
      iii) **IMO**: Select XTAL.
      iv) **USB**: Enable and s**ele**ct IMOx2 – 48.000 MHz.

   Note If the selected device is PSoC 3 ES2 or PSoC 5, you must also configure the PLL to "Desired 33 MHz" and the Master Clock to "PLL_OUT (33.000 MHz)."
4. Select **Build** to generate APIs.

Input/Output Connections

This section describes the various 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.

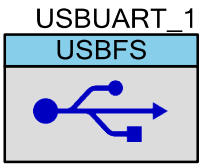
sof – Output *

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.

Schematic Macro Information

The PSoC Creator Component Catalog contains a Schematic Macro implementation of a 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 required configuration.

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.



Component Parameters

Drag a USBFS component onto your design and double-click it to open the **Configure USBFS** dialog.

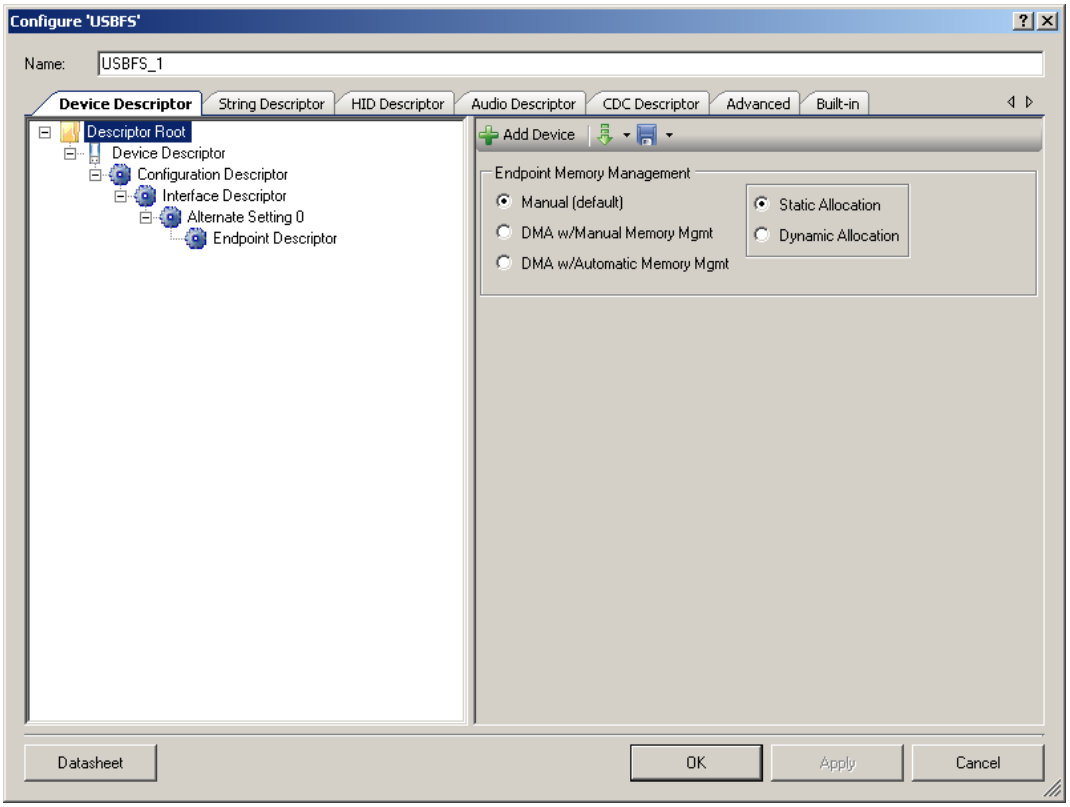
The component is driven by information generated by the USBFS Configure dialog. This dialog, or "customizer," facilitates the construction of the USB descriptors and integrates the information generated into the driver firmware used for device enumeration.

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.

The **Configure USBFS** dialog contains the following tabs and settings:

Device Descriptor Tab

Descriptor Root



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.

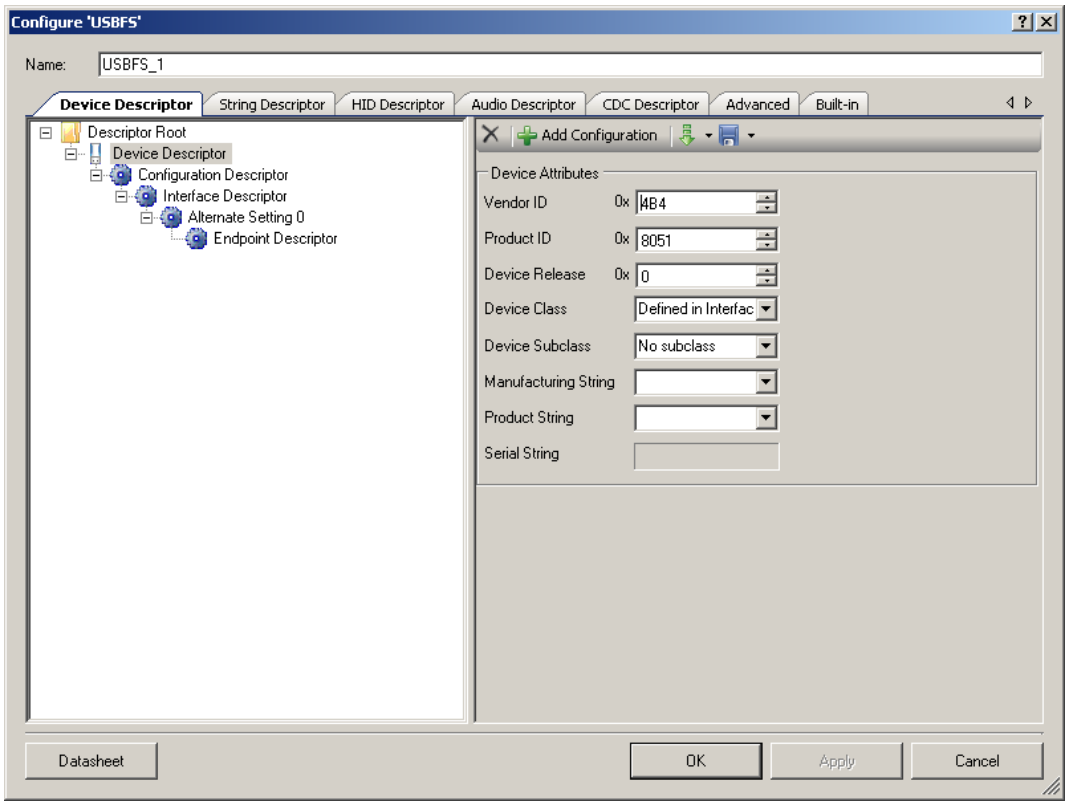
- **Manual** (default) – Select this option to use LoadInEP/ReadOutEP to load and unload the endpoint buffers.
  - **Static Allocation** – The memory for the endpoints is allocated immediately after a SET_CONFIGURATION request. This takes longest when multiple Alternate settings use the same endpoint (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. This option is supported for PSoC 3 Production silicon only.
- **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 LoadInEP/ReadOutEP functions for initial DMA configuration. This option is supported for PSoC 3 Production silicon only.

PSoC 3 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



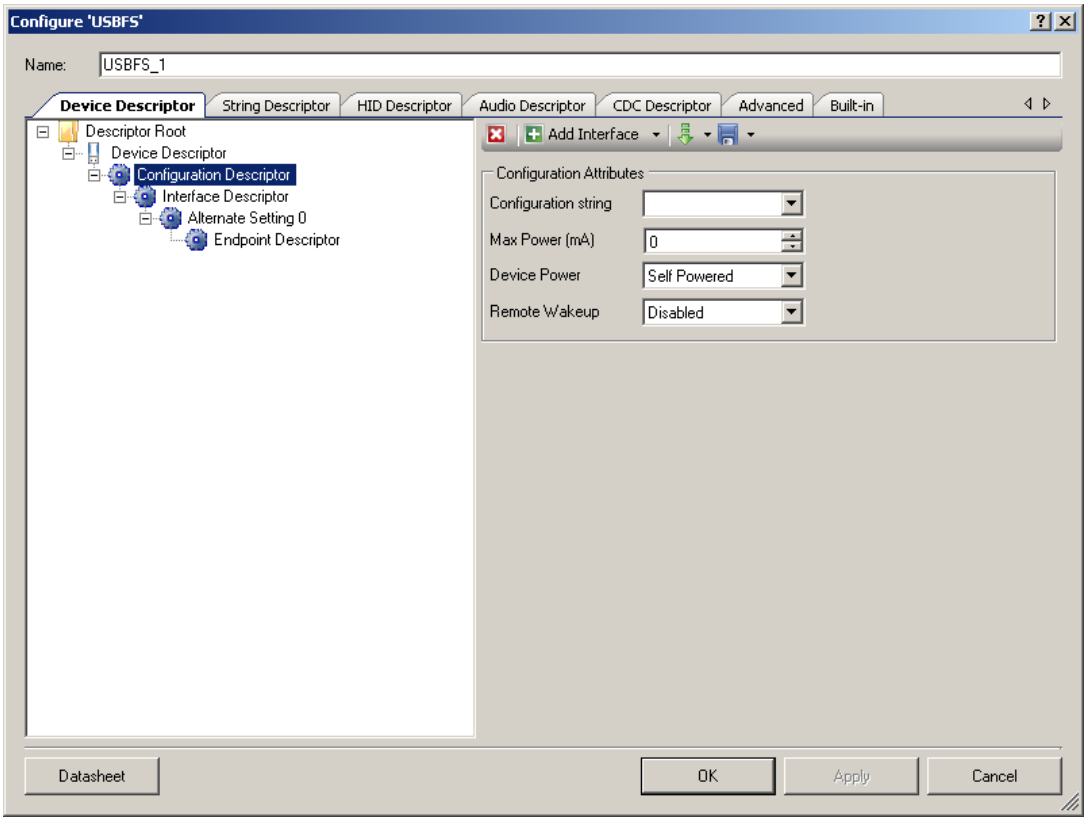
Device Attributes

- **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**
- **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**

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.

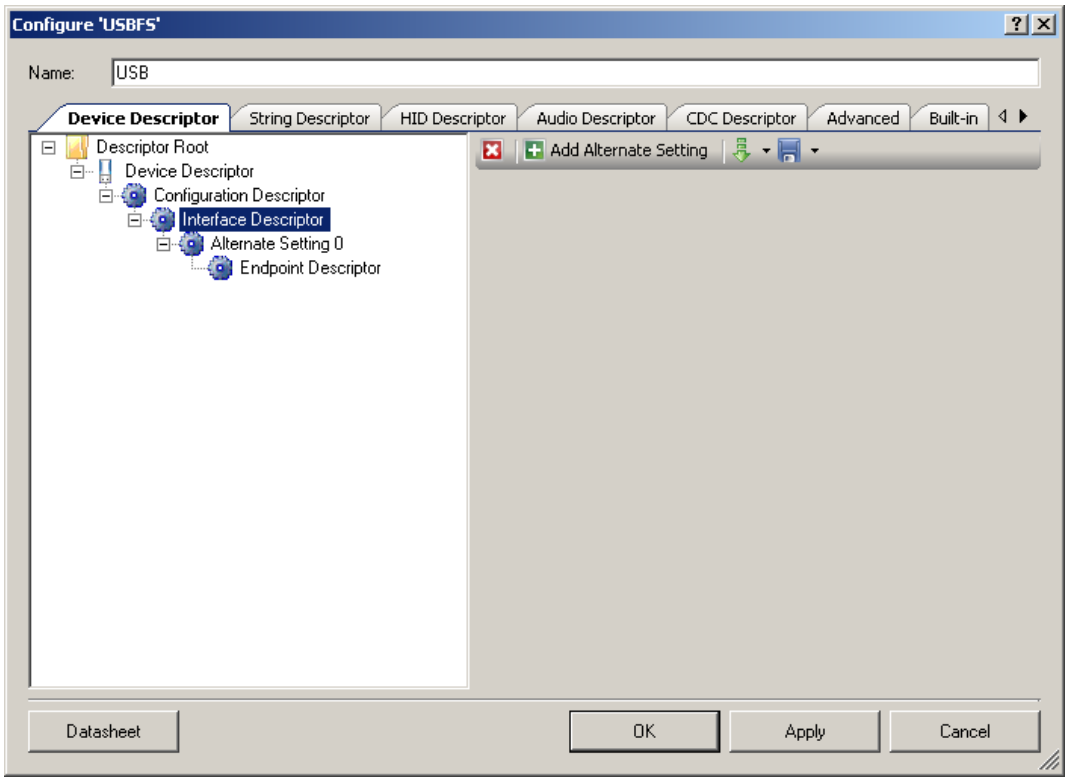
  Note The **Device Power** parameter reports whether the configuration is bus powered or self powered. Device status reports whether the device is currently self powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self powered. 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 Descriptor

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

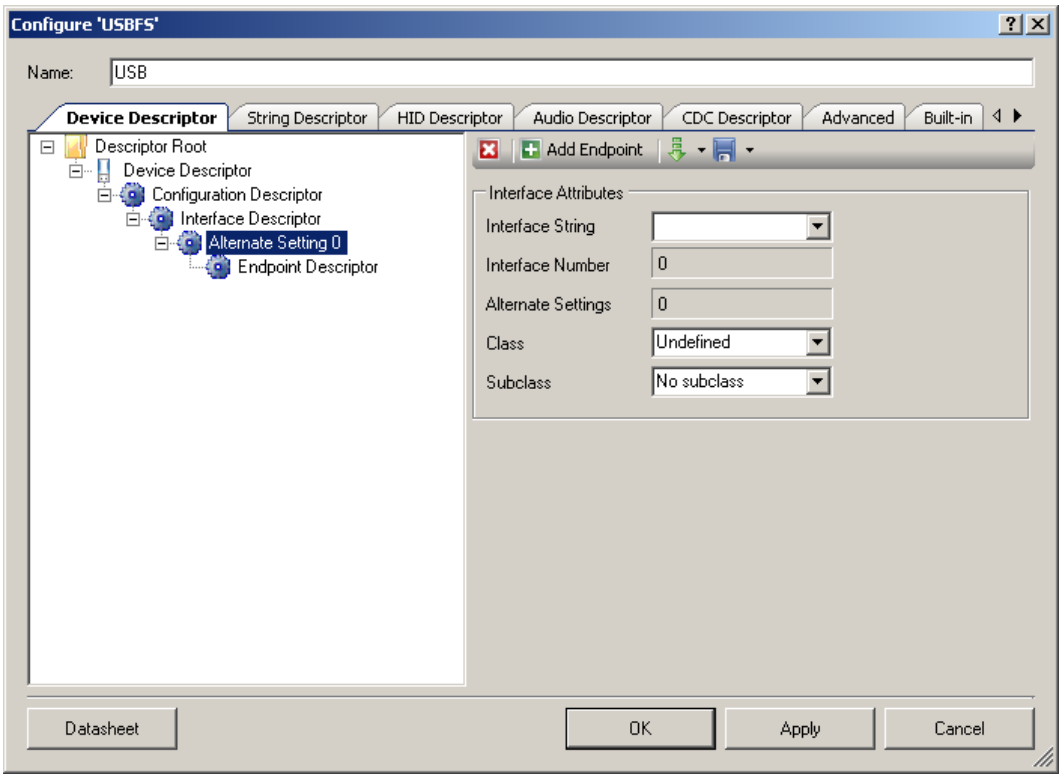


Alternate Setting 0 is automatically provided to configure your device. If your device uses isochronous endpoints, note that the USB 2.0 specification requires that no device default interface settings can include any isochronous endpoints with nonzero data payload sizes. This is specified using **Max 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 size, 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** – The interface number is computed by the customizer.
- **Alternate Settings** – The alternate setting is computed by the customizer.
- **Class** – **HID**, **Vendor Specific**, or **Undefined**
- **Subclass** – Dependent on the selected class

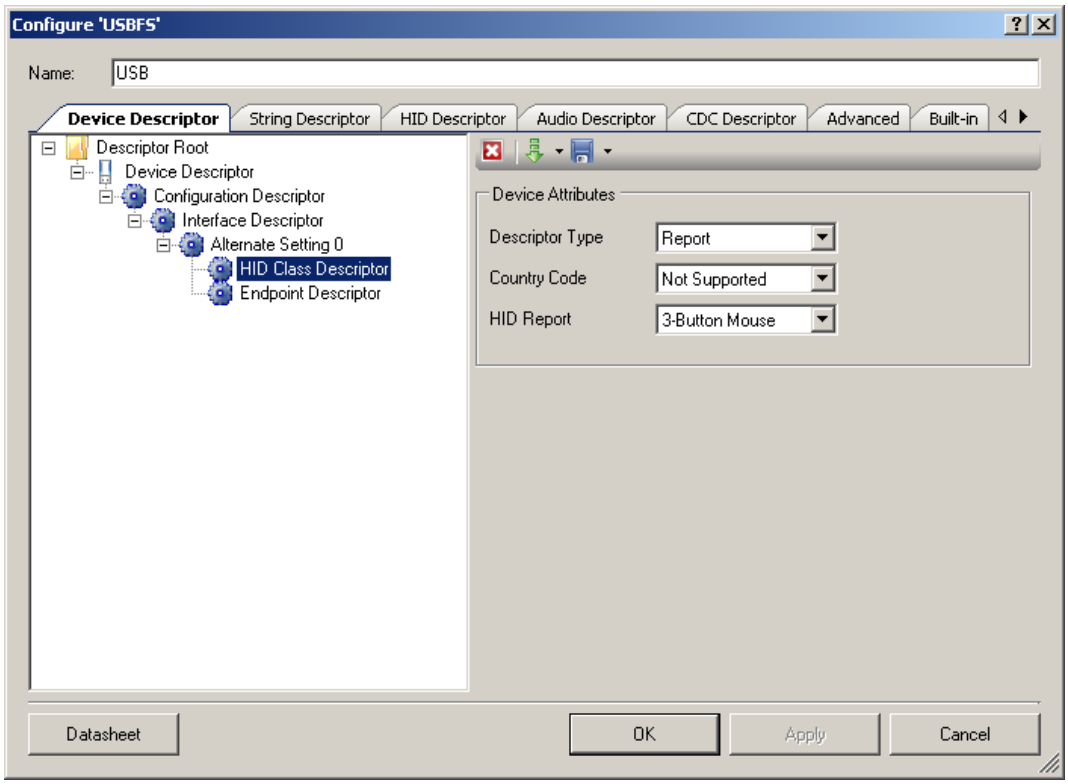
Note String descriptors are optional. If a device does not support string descriptors, all references to string descriptors within the device, configuration, and interface descriptors must be set to zero.

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

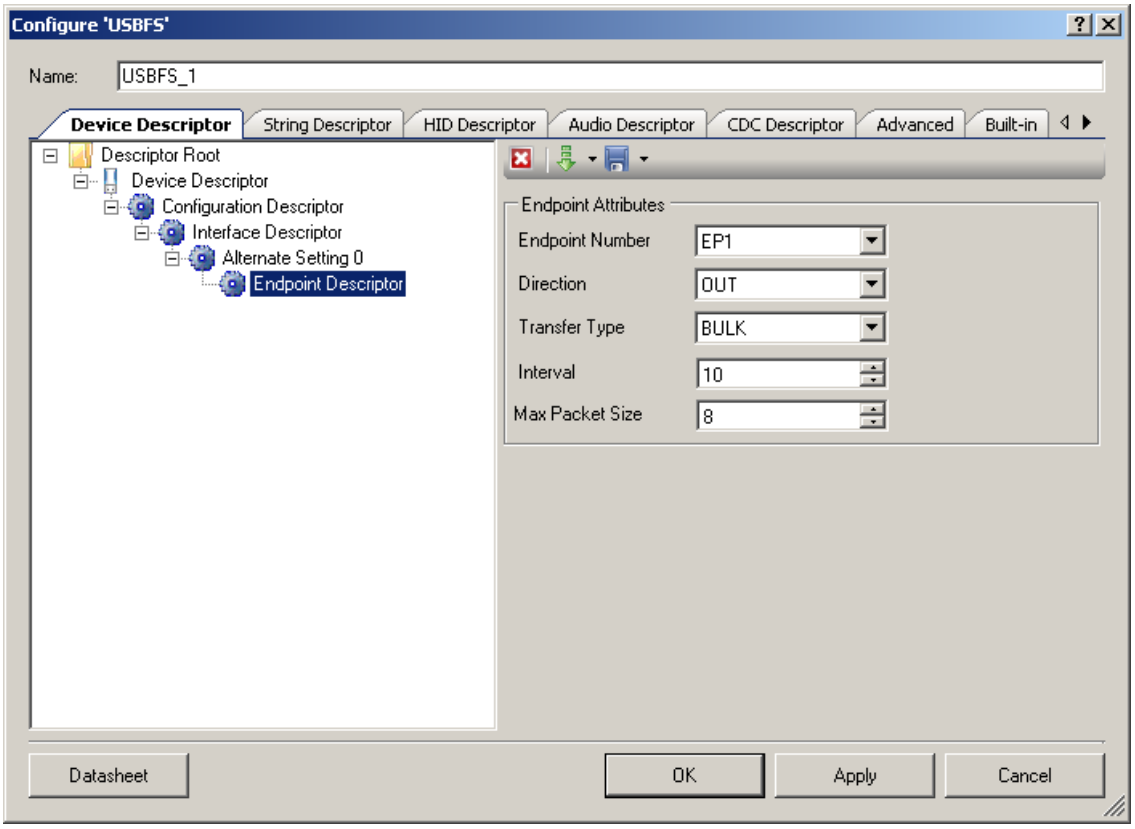
1. Select an **Alternate Setting** item in the **Descriptor Root** tree.
2. Under **Interface Attributes** on the right, select **HID** for the **Class** field.



Device 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.

  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.

Import and Save Tool Buttons

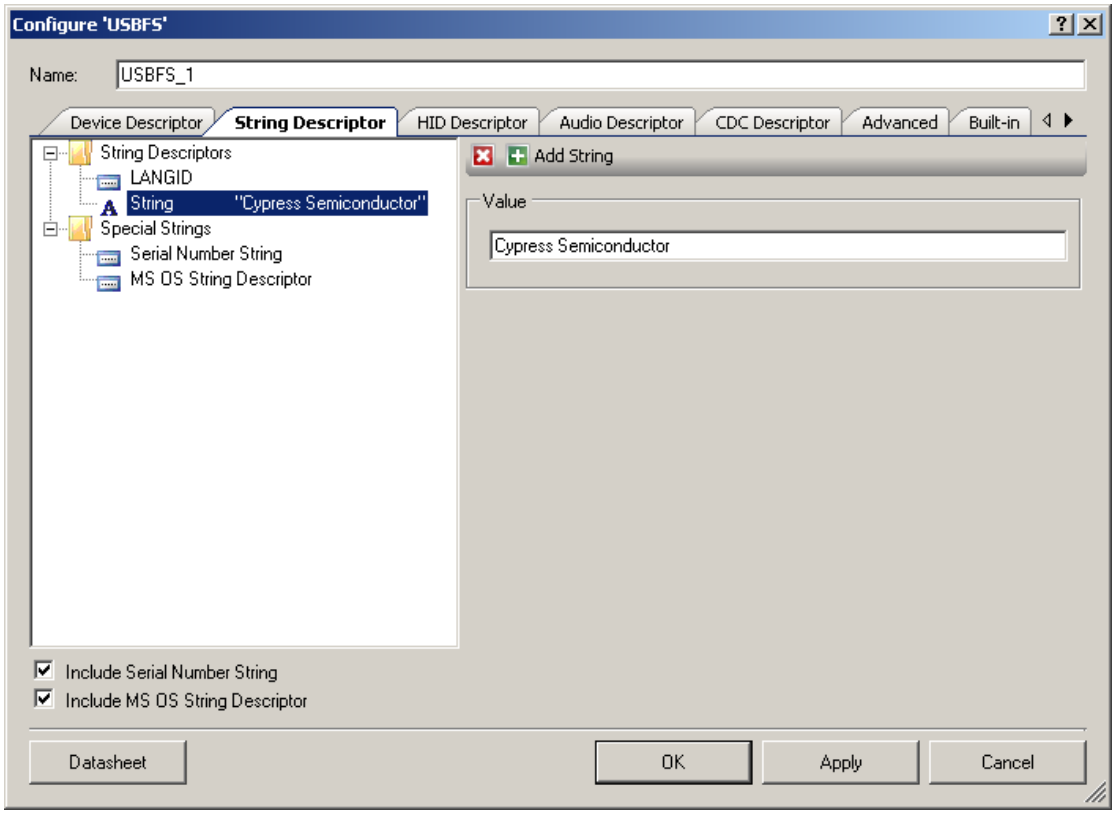
The **Save** button allows you to save information about the component configuration into an XML configuration file. In the drop-down list you can choose either **Save Current Descriptor** or **Save Root Descriptor**. The first option saves the configuration of the selected descriptor. The second option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

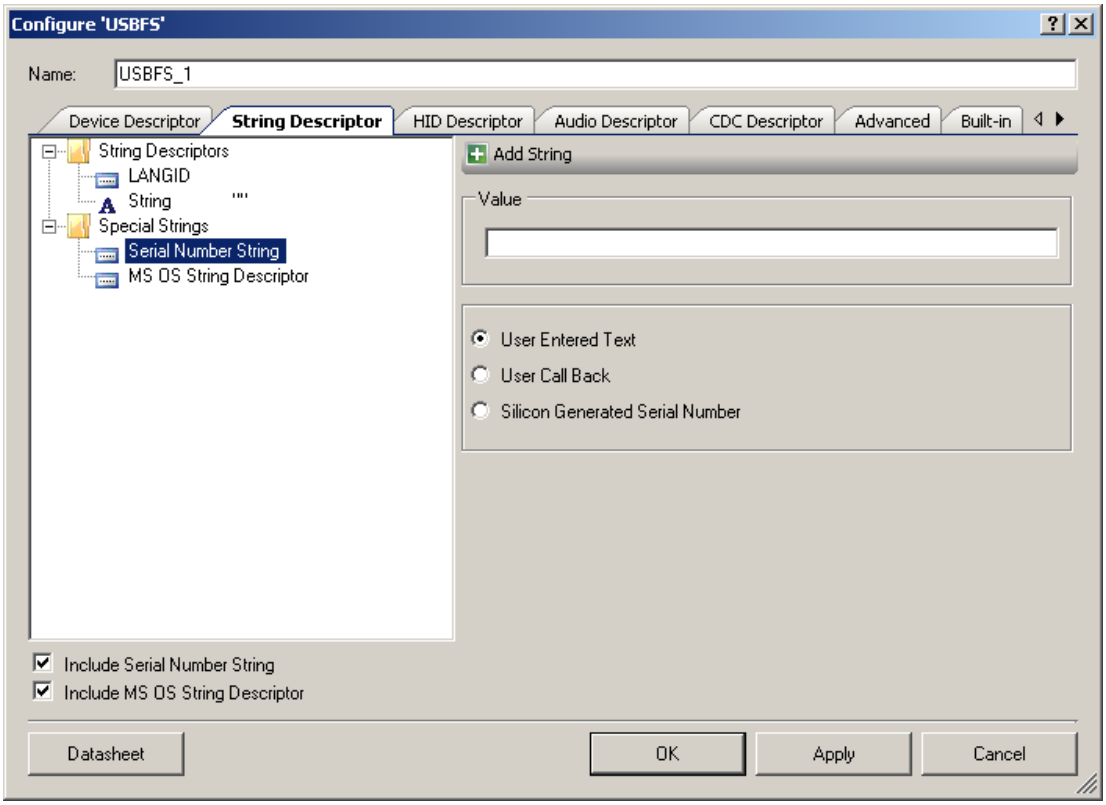
String Descriptor Tab

String Descriptors



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

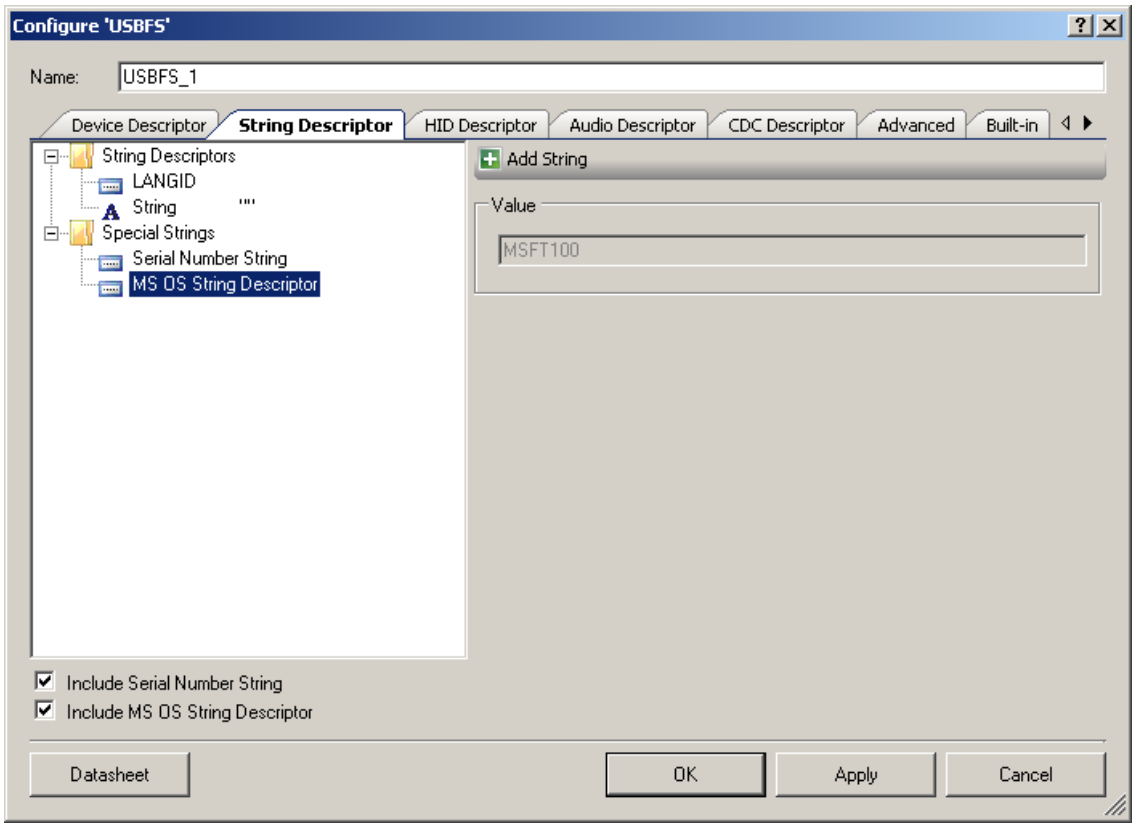
Serial Number String



- **Value** – Default string
- **User Entered Text** – Enables 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 runtime.
- **Silicon Generated Serial Number**

MS OS String Descriptor

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

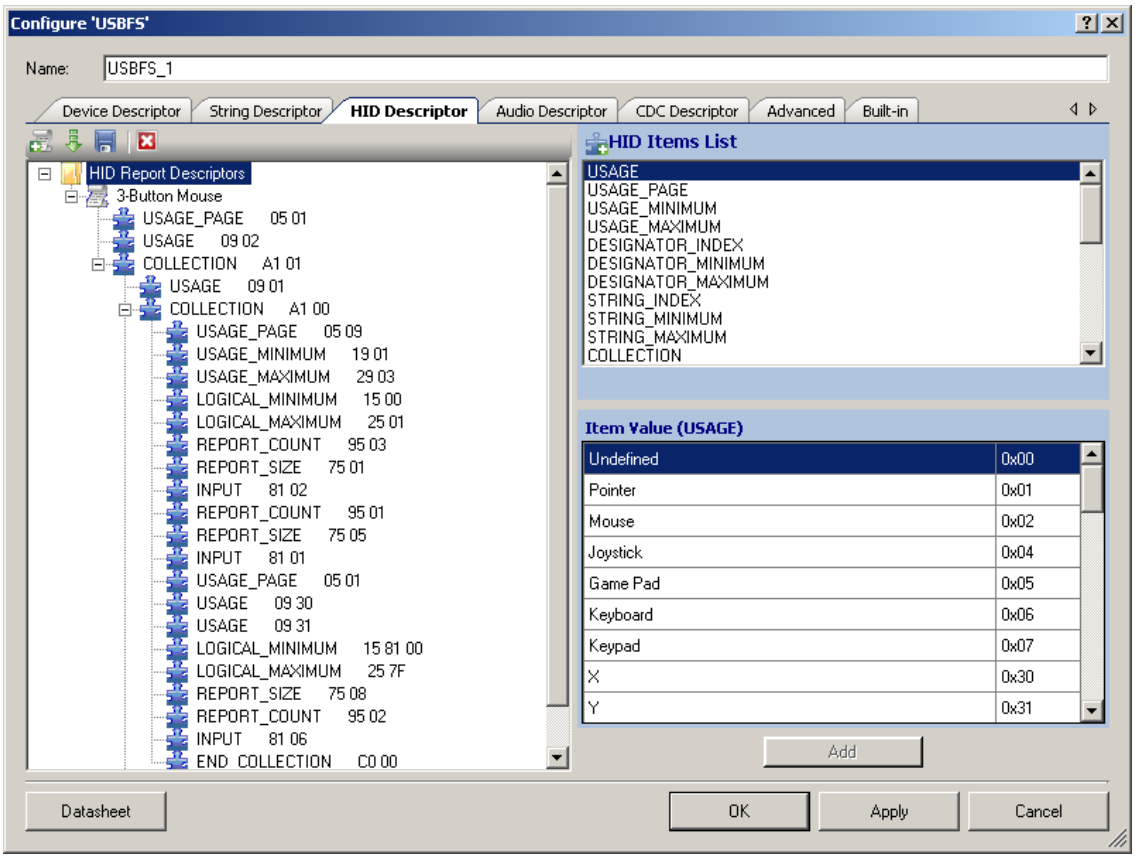


- **Value** – Constant string **MSFT100**

HID Descriptor Tab

The **HID Descriptor** tab allows you to quickly build HID descriptors for your device. Use the **Add Report** button to add and configure HID Report Descriptors.

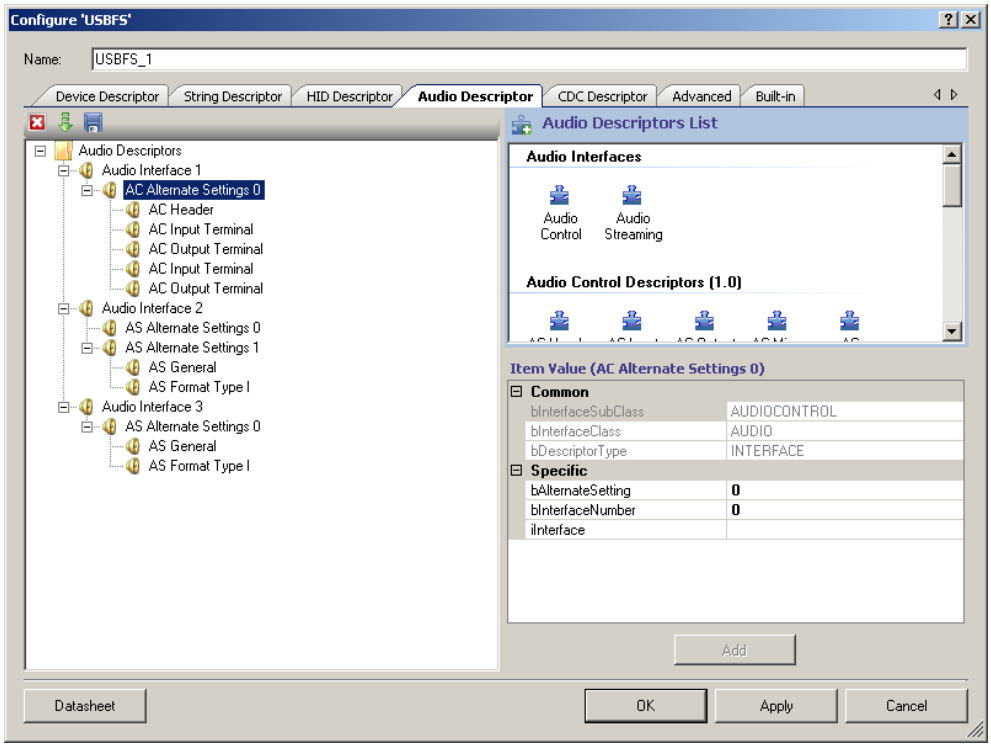
HID Descriptors



- **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.



To Add Audio Descriptors

1. Select the **Audio Descriptors** root item in the tree on the left.
2. Under **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 then 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 do 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

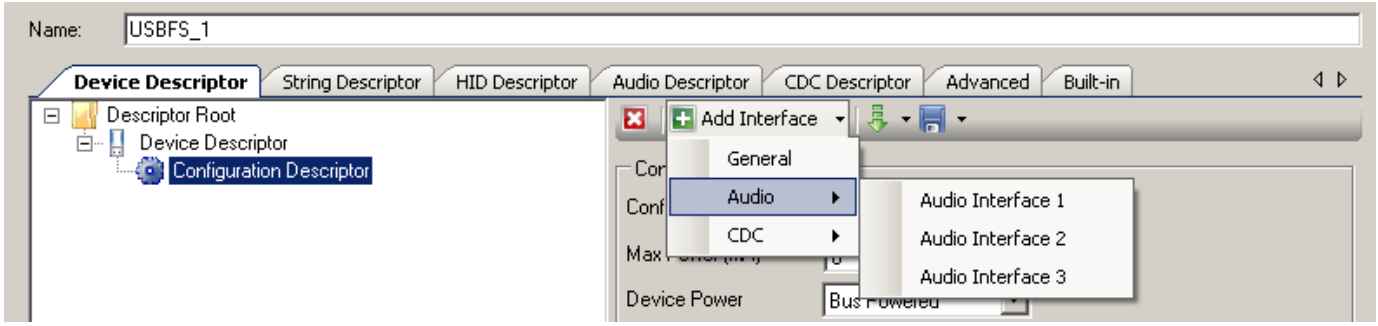
1. Select the appropriate **AC Alternate Settings x** or **AS Alternate Settings x** item in the tree on the left.
2. Under the **Audio Descriptors List** on the right, select the **Endpoint Descriptor** item.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **Add**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

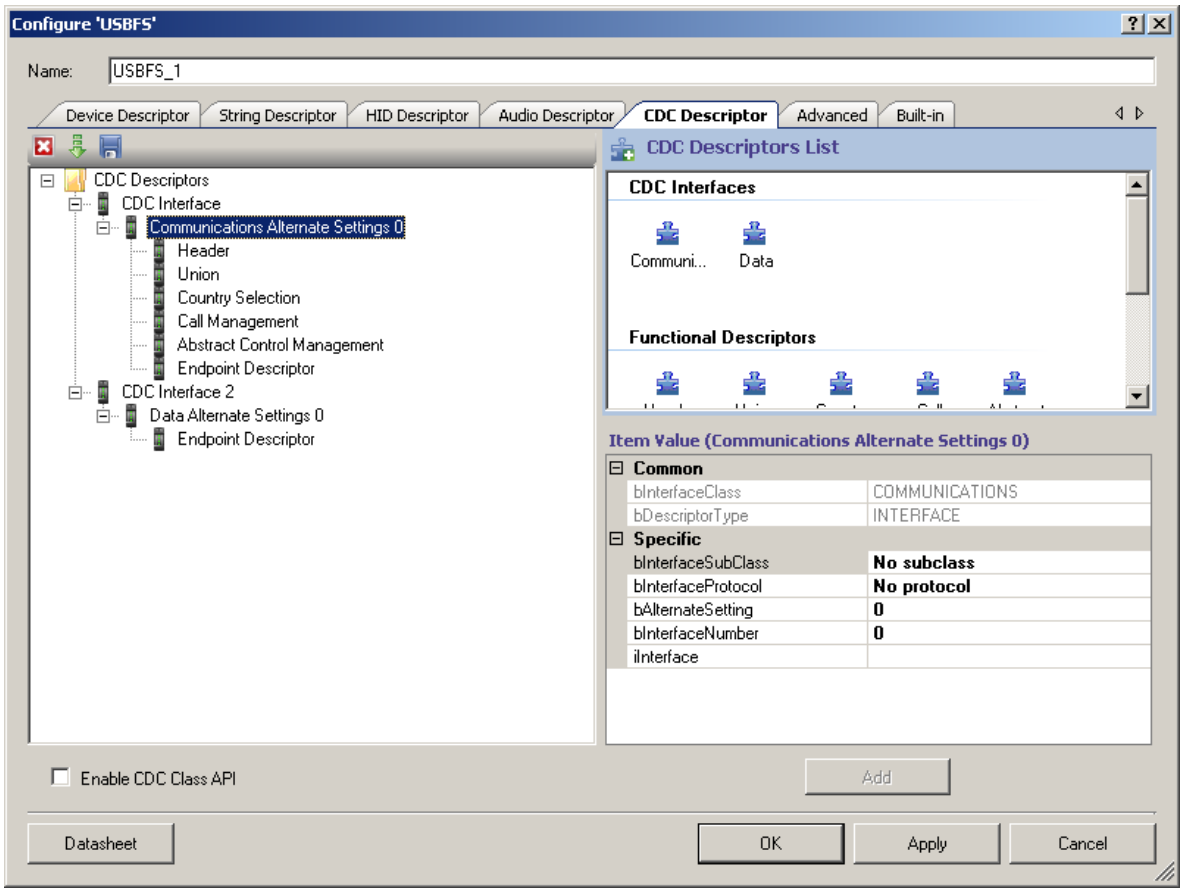


Audio interfaces are disabled in the **Device Descriptor** tab list because they can only be edited on the **Audio Descriptor** tab.

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

CDC Descriptor Tab

The **CDC Descriptor** tab is used to add and configure communications and data interface descriptors.



To Add CDC Descriptors

1. Select the **CDC Descriptors** root item in the tree on the left.
2. Under **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 then clicking on it.

To Add Functional Descriptors

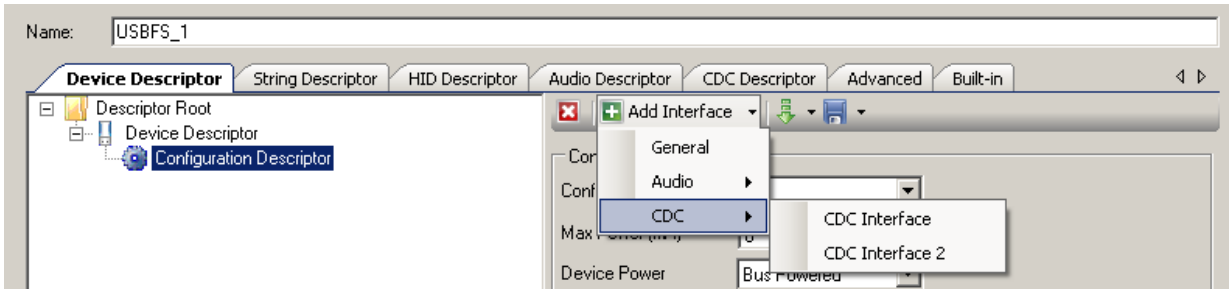
1. Select the appropriate **Communications Alternate Settings x** item in the tree on the left.
2. Under the **CDC Descriptors List** on the right, select one of the items under **Functional Descriptors** as appropriate.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click **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

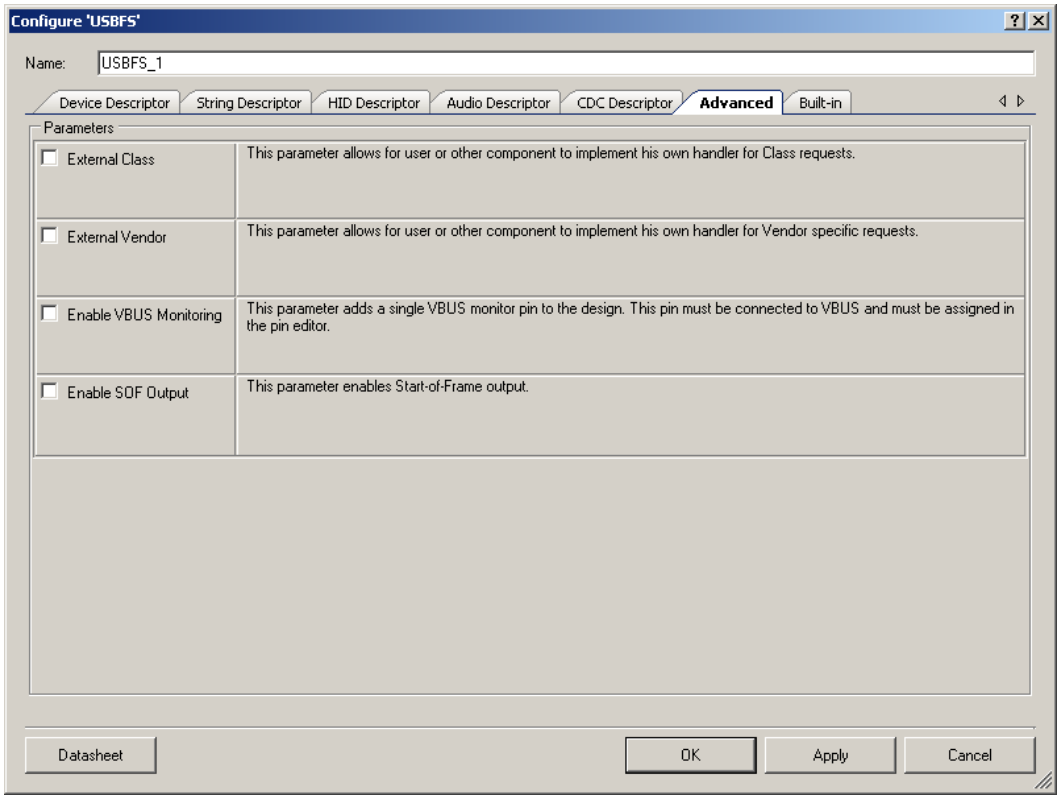
1. Go to the **Device Descriptor** tab.
2. Select the **Configuration Descriptor** to which a new interface will belong.
3. Click the **Add Interface** tool button, choose **CDC**, and select the appropriate item to add.



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**, all the descriptors you added will not be saved.

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. This pin must be connected to the VBUS and must be assigned in the Pin Editor. See the USB Compliance for Self-Powered Devices section for additional information.

Enable SOF Output

This parameter enables Start-of-Frame output.

Placement

USB is implemented as a fixed-function block.

Resources

Resources	Resource Type				API Memory (Bytes)		Pins (per External I/O)
	Clock Dividers	Macrocells	Interrupts	USB Fixed Blocks	Flash	RAM	
USBFS	0	0	7	1	5925	133	2

Clock Settings

The USB hardware block requires 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.
- If the selected device is PSoC 3 ES2 or PSoC 5, the Bus Clock cannot be slower than 33 MHz [for Master Clock, select PLL_OUT (33.000 MHz)].
- If the selected device is PSoC 5, the IMO must be sourced from the external 24-MHz XTAL.

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 for PSoC 3

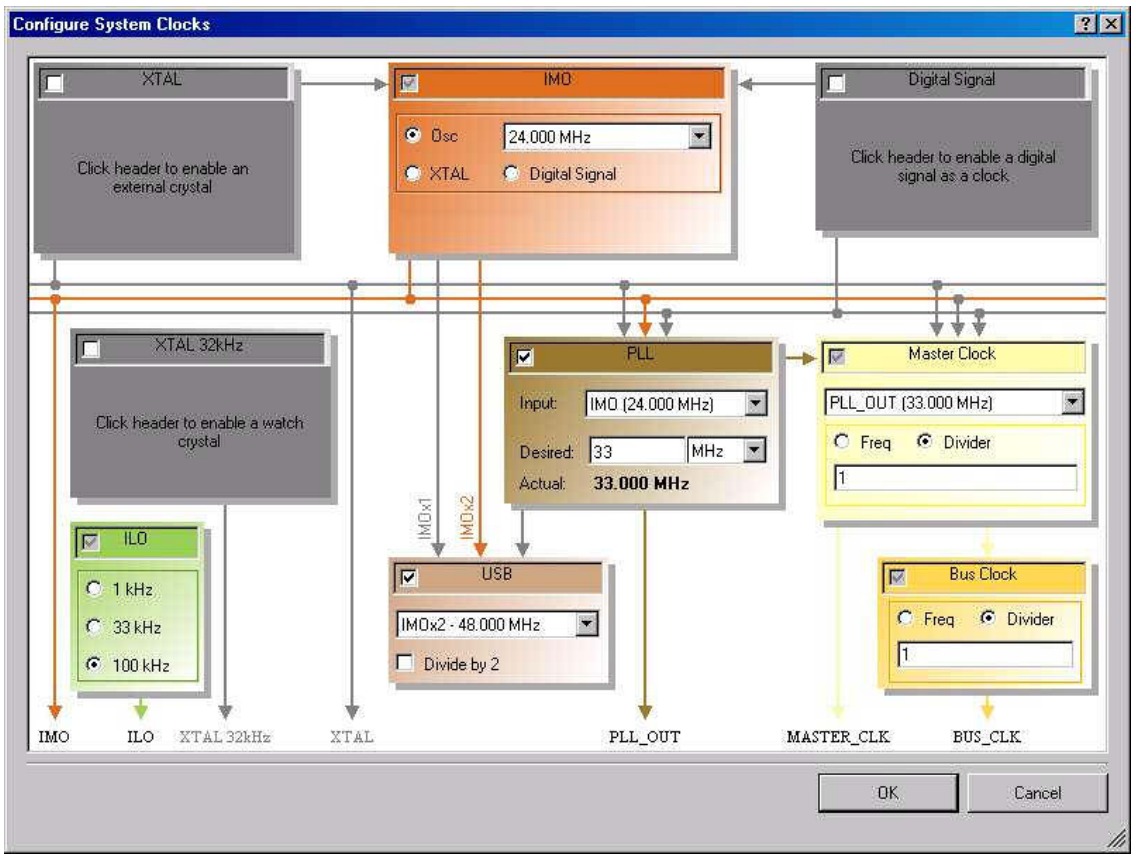
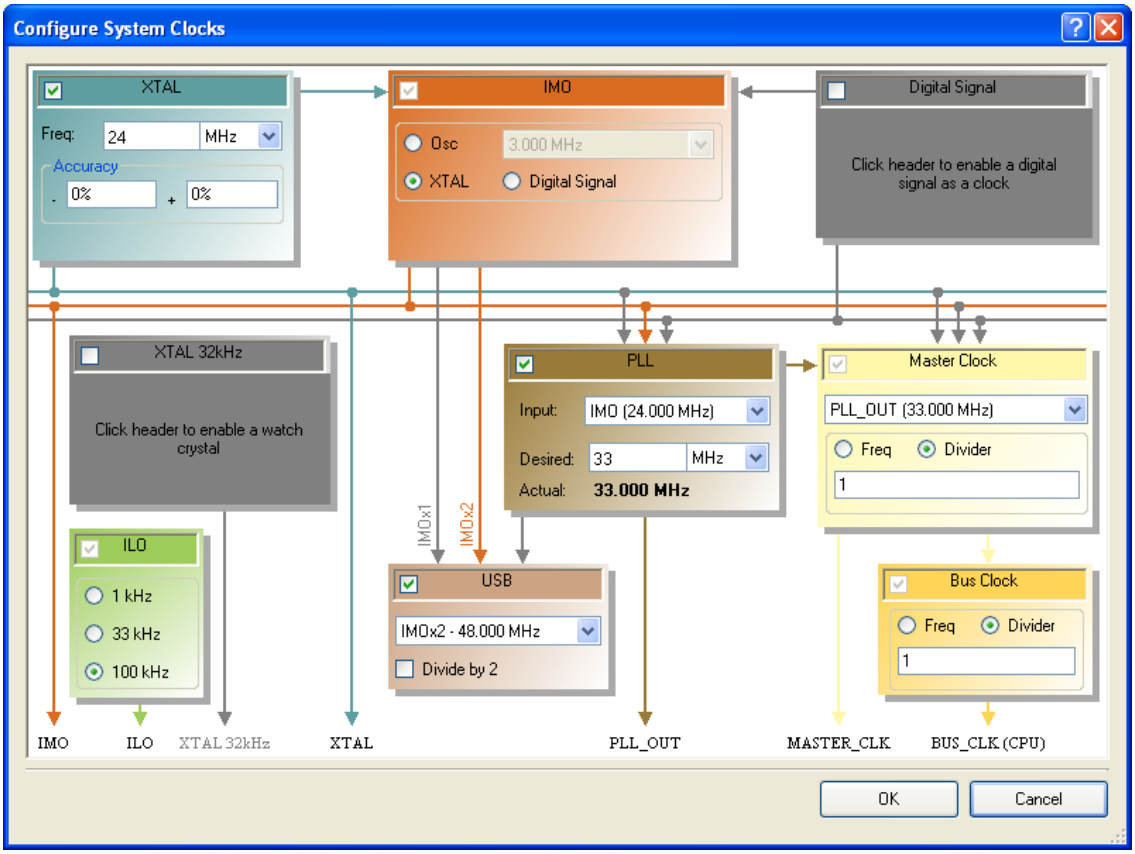


Figure 2. System Clock Configuration for PSoC 5



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 cover 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()	Identifies whether ACK was set by returning a nonzero value.
USBFS_GetEPCount()	Returns the current byte count from the specified USBFS endpoint.
USBFS_InitEP_DMA()	This function 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 runtime.
USBFS_TerminateEP()	Terminates endpoint transfers.

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 in 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 USBFS_Start() or USBFS_InitComponent() APIs.

Variable	Description
USBFS_transferState	This variable is used by the communication functions to handle current transfer state. Initialized to TRANS_STATE_IDLE in the USBFS_InitComponent() API and after complete transfer in status stage. Changed to the TRANS_STATE_CONTROL_READ or TRANS_STATE_CONTROL_WRITE in setup transaction depending on request type.
USBFS_configuration	Contains 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 returns 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_deviceAddress	Contains the current device address. This variable is initialized to zero in the USBFS_InitComponent() API. The host starts to communicate to the device with address 0 and then sets it to the value in the SET_ADDRESS request.
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: Performs all required initialization for the USBFS Component.

Parameters: uint8 device: Contains the device number from the appropriate device-descriptor set entered with the USBFS customizer.

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 voltage regulator and pass-through V_{CC} for pull-up
USBFS_5V_OPERATION	Enable voltage regulator and use regulator for pull-up
USBFS_DWR_VDDD_OPERATION	Enable or disable voltage regulator depend on V_{DDD} voltage configuration in DWR

Return Value: None

Side Effects: None

void USBFS_Init(void)

Description: 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: Initializes the component's global variables and initiate communication with the host by pull-up D+ line.

Parameters: uint8 device: Contains the device number from the appropriate device-descriptor set entered with the USBFS customizer.

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 voltage regulator and pass-through V_{CC} for pull-up
USBFS_5V_OPERATION	Enable voltage regulator and use regulator for pull-up
USBFS_DWR_VDDD_OPERATION	Enable or disable voltage regulator depend on V_{DDD} voltage configuration in DWR

Return Value: None

Side Effects: None

void USBFS_Stop(void)

Description: Performs all necessary shutdown tasks required for the USBFS component.

Parameters: None

Return Value: None

Side Effects: None

uint8 USBFS_GetConfiguration(void)

Description: 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: 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 nonzero value when new configuration has been changed; otherwise, returns zero.

Side Effects: None

uint8 USBFS_GetInterfaceSetting(uint8 interfaceNumber)

Description: 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: Returns the state of the requested endpoint.

Parameters: uint8 epNumber: The 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	Indicates that the endpoint is awaiting SIE action
USBFS_EVENT_PENDING	Indicates that the endpoint is awaiting CPU action
USBFS_NO_EVENT_ALLOWED	Indicates that 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: Determines whether or not an ACK transaction occurred on this endpoint by reading the ACK bit in the control register of the endpoint. This function does not clear the ACK bit.

Parameters: uint8 epNumber: Contains the data endpoint number.

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

Side Effects: None

uint16 USBFS_GetEPCount(uint8 epNumber)

Description: 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, 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.

uint8 *pData: A pointer to a data array that is related to the EP transfers.

Return Value: None

Side Effects: None

void USBFS_LoadInEP(uint8 epNumber, uint8 *pData, uint16 length)

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

Manual DMA:

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

Automatic DMA:

- Configures DMA. This is required only once, therefore it is done only when parameter pData is not NULL. When 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.

Parameters: uint8 epNumber: Contains the data endpoint number.

uint8 *pData: A 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: 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 USB_ReadOutEP() API and before expected data usage it is required to wait on DMA transfer 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: A 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: 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: 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: 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: Forces a USB J, K, or SE0 state on the D+/D– lines. This function provides the necessary mechanism for a USB device application to perform a USB Remote Wakeup. For more information, refer to 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 runtime. 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 user-defined string descriptor. String descriptor should meet the *Universal Serial Bus Specification revision 2.0* chapter 9.6.7

Return Value: None

Side Effects: None

void USBFS_TerminateEP(uint8 epNumber)

Description: 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: Device response NAK to the IN or OUT token to the terminated endpoint.

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

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.

uint8 USBFS_UpdateHIDTimer(uint8 interface)

Description: Updates the HID Report idle timer and returns the status. 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	Returned if the report is sent when data or state changes.

Side Effects: None

uint8 USBFS_GetProtocol(uint8 interface)

Description: 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_v1_60\Custom\template

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

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: 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: 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: Resets the receive and transmit communication buffers.

Parameters: None

Return Value: None

Side Effects: None

cystatus USBFS_CyBtldrCommWrite(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)

Description: 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.

Parameters: uint8 *data: 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 refer to the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

cystatus USBFS_CyBtldrCommRead(uint8 *data, uint16 size, uint16 *count, uint8 timeOut)

Description: 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.

Parameters: uint8 *data: 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 refer to 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 wake up status.

uint8 USBFS_CheckActivity(void)

Description: Returns the activity status of the bus. 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	If bus activity was detected since the last call to this function
0	If bus activity was not detected since the last call to this function

Side Effects: None

void USBFS_Suspend(void)

Description: Disables the USBFS block and prepares for power down mode. Should be called just before entering sleep.

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:

void USBFS_Resume(void)

Description: Enables the USBFS block after power down mode. 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. 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:

uint8 USBFS_RWUEnabled(void)

Description: Returns current remote wakeup status.

If the device supports remote wakeup, the application can determine if the host enabled remote wakeup with the USBFS_RWUEnabled() API function. 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:

Audio Class Support

Global Variables

Variable	Description
USBFS_currentSampleFrequency	Contains the current audio Sample Frequency. It is set by the host using SET_CUR request to the endpoint.

Variable	Description

USBFS_frequencyChanged	This variable is 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;
	}
	USBFS_transferState variable is checked to be sure that transfer completes.

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

CDC Class 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 w/Automatic Memory Mgmt.

Function	Description
USBFS_CDC_Init()	Initializes the CDC interface to be ready for the receive data from the PC
USBFS_PutData()	Sends a specified number of bytes from the location specified by a pointer to the PC
USBFS_PutString()	Sends a null terminated string to the PC
USBFS_PutChar()	Writes a single character to the PC
USBFS_PutCRLF()	Sends a carriage return (0x0D) and line feed (0x0A) to the PC
USBFS_GetCount()	Returns the number of bytes that were received from the PC
USBFS_CDCIsReady()	Returns a nonzero value if the component received data
USBFS_DataIsReady()	Returns a nonzero value if the component is ready to send more data
USBFS_GetData()	Gets a specified number of bytes from the input buffer and places them in a data array specified by the passed pointer
USBFS_GetAll()	Gets all bytes of received data from the input buffer and places them into a specified data array
USBFS_GetChar()	Reads one byte of received data from the buffer
USBFS_IsLineChanged()	Returns clear-on-read status of the line
USBFS_GetDTERate()	Returns the data terminal rate set for this port in bits per second
USBFS_GetCharFormat()	Returns the number of stop bits

Function	Description
USBFS_GetParityType()	Returns the parity type for the CDC port
USBFS_GetDataBits()	Returns the number of data bits for the CDC port
USBFS_GetLineControl()	Returns line control bitmap

Global Variables

Variable	Description
USBFS_lineCoding	Contains the current line coding structure. It is set by the host using a SET_LINE_CODING request and returned to the user code by the USBFS_GetDTERate(), USBFS_GetCharFormat(), USBFS_GetParityType(), and USBFS_GetDataBits() APIs.
USBFS_lineControlBitmap	Contains the current control-signal bitmap. It is set by the host using a SET_CONTROL_LINE request and returned to the user code by the USBFS_GetLineControl() API.
USBFS_lineChanged	Used as a flag for the USBFS_IsLineChanged() API, to inform it that the host has been sent a request to change line coding or control bitmap.
USBFS_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.
USBFS_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 USBFS_CDC_Init(void)

Description: Initializes the CDC interface to be ready for the receive data from the PC. This API should be called after the device has been configured.

Parameters: None.

Return Value: None.

Side Effects: None

void USBFS_PutData(uint8* pData, uint16 length)

Description: Sends a specified number of bytes from the location specified by a pointer to the PC.

Parameters: pData: Pointer to the buffer containing data to be sent.
length: Specifies the number of bytes to send from the pData buffer. Maximum length is limited by the maximum packet size for the endpoint.

Return Value: None

Side Effects: None

void USBFS_PutString(char8* string)

Description: Sends a null terminated string to the PC.

Parameters: string: Pointer to the string to be sent to the PC.

Return Value: None

Side Effects: None

void USBFS_PutChar(char8 txDataByte)

Description: Writes a single character to the PC.

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

Return Value: None

Side Effects: None

void USBFS_PutCRLF(void)

Description: Sends a carriage return (0x0D) and line feed (0x0A) to the PC.

Parameters: None

Return Value: None

Side Effects: None

uint16 USBFS_GetCount(void)

Description: Returns the number of bytes that were received from the PC.

Parameters: None

Return Value: uint16: Returns the number of received bytes.

Side Effects: None

uint8 USBFS_DataIsReady(void)

Description: Returns a nonzero value if the component received data or received a zero-length packet. The USBFS_GetAll() or USBFS_GetData() API should be called to read data from the buffer and reinitialize the OUT endpoint even when a 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 USBFS_CDCIsReady(void)

Description: Returns a nonzero value if the component is ready to send more data to the PC. Otherwise returns zero. Should be called before sending new data to ensure that the previous data has finished sending.

Parameters: None

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

Side Effects: None

uint16 USBFS_GetData(uint8* pData, uint16 length)

Description: Gets a specified number of bytes from the input buffer and places them in a data array specified by the passed pointer. The USBFS_DataIsReady() API should be called first, to be sure that data is received from the host.

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

length: Number of bytes to read into the data array from the RX buffer. Maximum length is limited by the number of received bytes.

Return Value: uint16: Number of bytes received.

Side Effects: None

uint16 USBFS_GetAll(uint8* pData)

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

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

Return Value: uint16: Number of bytes received.

Side Effects: None

uint8 USBFS_GetChar(void)

Description: Reads one byte of received data from the buffer.

Parameters: None

Return Value: uint8: Received one character.

Side Effects: None

uint8 USBFS_IsLineChanged(void)

Description: This function returns clear-on-read status of the line.

Parameters: None

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

Return Value	Description
USBFS_LINE_CODING_CHANGED	Line coding changed
USBFS_LINE_CONTROL_CHANGED	Line control changed

Side Effects: None

uint32 USBFS_GetDTERate(void)

Description: 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 USBFS_GetCharFormat(void)

Description: Returns the number of stop bits.

Parameters: None

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

Return Value	Description
USBFS_1_STOPBIT	1 stop bit
USBFS_1_5_STOPBITS	1,5 stop bits
USBFS_2_STOPBITS	2 stop bits

Side Effects: None

uint8 USBFS_GetParityType(void)

Description: Returns the parity type for the CDC port.

Parameters: None

Return Value: uint8:

Return Value	Description
USBFS_PARITY_NONE	None
USBFS_PARITY_ODD	Odd
USBFS_PARITY_EVEN	Even
USBFS_PARITY_MARK	Mark
USBFS_PARITY_SPACE	Space

Side Effects: None

uint8 USBFS_GetDataBits(void)

Description: 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 USBFS_GetLineControl (void)

Description: Returns Line control bitmap.

Parameters: None.

Return Value: uint8:

Return Value	Notes
USBFS_LINE_CONTROL_DTR	Indicates DTR signal is present.. This signal corresponds to V.24 signal 108/2 and RS232 signal DTR.
USBFS_LINE_CONTROL_RTS	Carrier control for half-duplex modems. This signal corresponds to V.24 signal 105 and RS232 signal RTS.

Side Effects: None

Code Example (CE60246) USBUART Migration

Before the addition of the 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 on 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 above).

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(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

*Table assumes component name is "USBUART_1"

Interrupt Service Routine

Empty SOF ISR is provided with this component. It is not enabled by default. If your application requires this interrupt it can be enabled by calling:

CyIntEnable(USBFS_SOF_VECT_NUM);

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

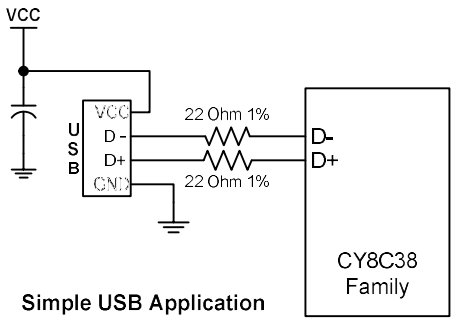
Sample Firmware Source Code

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.

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
GET_DESCRIPTOR	Returns the specified descriptor	9.4.3
GET_INTERFACE	Returns the selected alternate interface setting for the specified interface	9.4.4
GET_STATUS	Device	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

Class Request	USBFS Component Support Description	Device Class Definition for Audio - Section
GET_CUR	Interface: MUTE_CONTROL VOLUME_CONTROL	5.2.1.1
	Endpoint: SAMPLING_FREQ_CONTROL	
SET_CUR	Interface: MUTE_CONTROL VOLUME_CONTROL	5.2.1.2
	Endpoint: SAMPLING_FREQ_CONTROL	

CDC Class Request

Class Request	USBFS Component Support Description	Communications Class Subclass Specification for PSTN Devices
SET_LINE_CODING	Allows the host to specify typical asynchronous line-character formatting properties. 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	6.3.12

DC and AC Electrical Characteristics

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	–	7	10	mA
		V_{DDD} = 3.3 V	–	5	8	mA
$I_{USB_Suspended}$	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	1.2	mA
		V_{DDD} = 5 V, disconnected from USB host	–	0.3	1.0	mA
		V_{DDD} = 3.3 V, connected to USB host, PICU configured to wake on USB resume signal	–	0.5	1.2	mA
		V_{DDD} = 3.3 V, disconnected from USB host	–	0.3	1.0	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
2.10	Fixed handling of the class-specific requests in USBFS_DispatchClassRqst() function.	The Audio requests were stalled.
2.0	Added CDC class support: ▪ New tab “CDC Descriptor” added. This tab lets user 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 two new groups of available descriptors were added. 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.
	Endpoint memory management group box was moved 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.
	Initialize USBFS_hidProtocol variable to HID_PROTOCOL_REPORT value in USBFS_InitComponent() and USBFS_reInitComonent() functions.	To comply with HID “7.2.6 Set_Protocol Request” --- “When initialized, all devices default to report protocol.”
	Added support 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 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”
	Boot loader APIs were renamed to have instance name ahead. The backward compatible defines were added.	Preparation for future ability to boot from multiple interfaces.

Version	Description of Changes	Reason for Changes / Impact
	Added characterization data to datasheet	
	Minor datasheet edits and updates	
1.50.a	Made datasheet change log cumulative	Customer convenience.
1.50	USB Suspend, Resume, and Remote Wakeup functionality has been added.	USB device should support suspend and resume functionality.
	Most APIs renamed to remove Hungarian notation, old names are supported for backward compatibility.	To comply with corporate coding standards.
	GET_INTERFACE/SET_INTERFACE requests support has been added.	A device must support the GetInterface/SetInterface requests if it has alternate settings for that interface.
	Specific APIs were integrated to support the boot loader: 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.
	'Advanced' tab layout modified.	The data grid was replaced with check boxes with information about each parameter. This was done 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 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.

Version	Description of Changes	Reason for Changes / Impact
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 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.
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, 2011. 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.