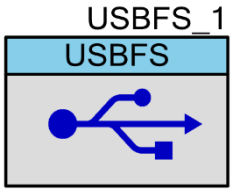
Full Speed USB (USBFS)

3.0

Features



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

General Description

The USBFS component provides a USB full-speed, Chapter 9 compliant device framework for constructing HID-based and generic USB devices. It provides a low-level driver for the control endpoint that decodes and dispatches requests from the USB host. Additionally, the component provides a GUI-based configuration dialog to aid in constructing your descriptors, allowing full device definition that can be imported and exported. Commonly used descriptor templates are provided with the component and can be imported as needed in your design.

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

The USBFS component has stringent clock requirements. The following conditions must be met for a valid USB design:

- The USB clock must be 48 MHz.
- The accuracy of the USB clock must be within +/-0.25%.

To meet these conditions, navigate to the **Clocks** tab in the Design-Wide Resources (DWR) file (*project.cydwr*). You may also drag and drop a USBFS component from the Component Catalog and then click on the error in the Notice List window. This will open the System Clock Editor in the DWR file.

Perform the following changes to use the internal PSoC clocks to drive the USB. Alternatively, use an external clock that satisfies the above conditions.

PSoC 4200L

1. IMO at 48 MHz, trim with USB.
    1. **IMO**: Set to 48 MHz.
    2. **Trim**: Set “Trim with” parameter in the IMO box as USB. You may also choose to trim with WCO.
2. IMO at 24 MHz, PLL at 48 MHz.
    1. **IMO**: Set to 24 MHz.
    2. **Trim**: Set “Trim with” parameter in the IMO box as USB. You may also choose to trim with WCO.
    3. **PLL0** or **PLL1**: Set either one of these to 48 MHz.
    4. **HFCLK**: Set to either PLL0 or PLL1 depending on the previous step.

PSoC 3 and PSoC 5LP

IMO at 24 MHz, doubler for USB.

1. **IMO**: Set to 24 MHz.
2. **ILO**: Set to 100 kHz
3. **USB**: Enable and select IMOx2 to achieve 48 MHz.

Once the clock configuration meets the requirements, no warnings or errors should appear in the Notice List window. Click on **Build** to generate your APIs.

An Introduction to Universal Serial Bus 2.0

Application Note AN57294 is a foundation for understanding the USB protocol, specifically focusing on the USB 2.0 specification. It is intended for those who are new to using USB in embedded designs, as well as for those who need to use and understand more advanced Cypress application notes.

Definitions

- USBFS – USB device supporting the full speed (FS) transfer mode
- ACK – USB Handshake packet indicating a positive acknowledgment
- CDC – Communication Device Class
- Descriptor – Data structure with a defined format used by USB devices to report their attributes to a USB host.
- DMA – Direct Memory Access
- Endpoint – A uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device.
- FAT – File Allocation Table
- HID – Human Interface Device
- ISR – Interrupt Service Routine
- LPM – Link Power Management
- MIDI – Musical Instrument Digital Interface
- MIDI Channel – A MIDI Channel is a bus over which devices sending or receiving MIDI data can communicate.
- MIDI Port – The point or points on a MIDI device where you connect to other MIDI devices.
- NAK – USB Handshake packet indicating a negative acknowledgment
- SIE – Serial Interface Engine
- SOF – Start of Frame
- USB – Universal Serial Bus
- USBUART – Universal Serial Bus Universal Asynchronous Receive Transmit macro

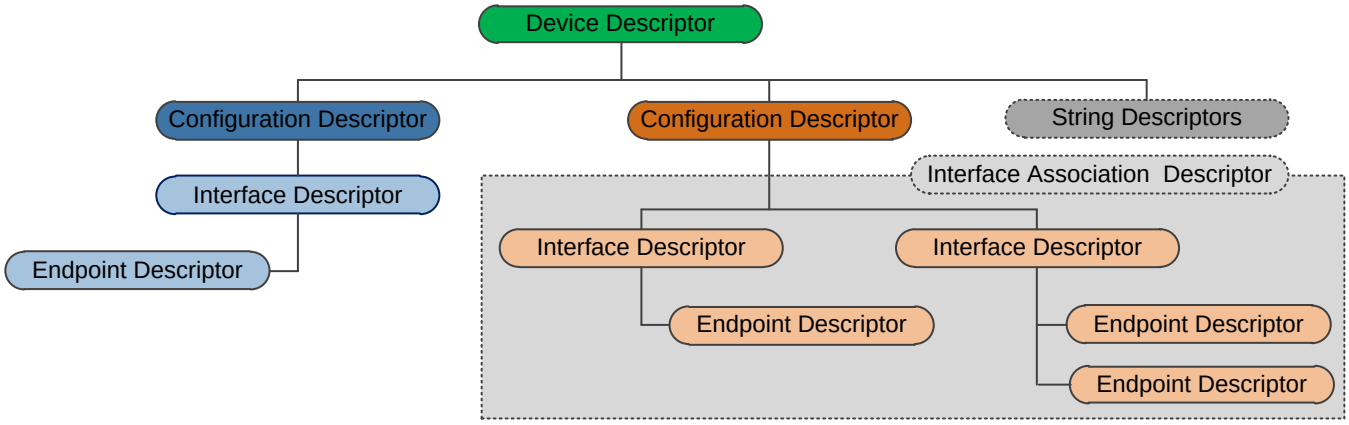
- USBMIDI – Universal Serial Bus Musical Instrument Digital Interface macro
- USB-IF – Universal Serial Bus Implementers Forum

USB Descriptor Configuration

The USBFS component provides a GUI-based framework, allowing you to design a USB Full Speed device by configuring the descriptors needed for your application. Descriptors are used by the device to inform the host of its configuration, type, capabilities and power requirements. This information is typically conveyed using a descriptor table as part of the device firmware. The USBFS component provides access to the 5 common USB descriptors, as well as some other miscellaneous descriptor types. The following subsections give an overview of each of these descriptors.

If you intend to design a USBFS Audio, USBFS MIDI, USBUART (CDC), or USBFS MSC device, then refer to the appropriate sections. Once the descriptors and component parameters are configured, you may design your application firmware using the provided APIs.

Figure 1. Device Descriptor Tree



Device Descriptor

Device descriptors are used to describe the characteristics of the device such as its product and vendor IDs, the device class, number of device configurations and protocols. There can be only one instance of the device descriptor per device, and this is defined for the entire device.

The USBFS component provides device descriptor fields in the **Device Descriptor** tab. You may add multiple devices in your design, which allows you to define a device descriptor for each device.

Configuration Descriptor

Configuration descriptors give information about a specific device configuration such as whether the device is self-powered or bus-powered, the maximum power (current draw), the number of interfaces and remote wakeup capability. When the device is enumerated, the host reads the configuration descriptors for that device, and chooses which configuration to enable (only one may be enabled at a time). For example a device may have a bus-powered configuration descriptor, and a self-powered configuration descriptor. The host will check for each of these configurations and enable whichever it is applicable for that particular host.

The USBFS component provides access to configuration descriptor fields in the **Device Descriptor** tab. You may instantiate multiple Configuration Descriptor instances in a device.

Interface Descriptor

Interface descriptors describe a specific interface within a configuration, which is a collection of Endpoint descriptors (EP1 to EP8) that are grouped to specify a certain function or feature. Each interface descriptor declares the USB class of the device, which identifies the device functionality and aids in the loading of a proper driver for that specific functionality. Multiple interfaces may be enabled at the same time, which enables multiple functionality for that single device enumeration.

The USBFS component provides access to interface descriptor fields in the **Device Descriptor** tab. You may instantiate multiple interface descriptor instances under a configuration. For multiple interface functionality, the **Interface Association Descriptor** should be used in conjunction with individual interface descriptors.

Interface Association Descriptor

The Interface Association Descriptor (IAD) is used to describe two or more interfaces that are associated with a single device function. It informs the host that the referenced interfaces are linked together. For example, a USB to UART bridge design has two interfaces associated with it: a control interface and a data interface. The IAD tells the host that these two interfaces are part of the same function, which is a USB-UART, and falls under the communication device class (CDC). This descriptor is not required in all cases of multiple interface designs. If an interface should have different functionality from another interface then they should remain separate.

The Interface Association Descriptor can be added under the configuration descriptor in the **Device Descriptor** tab in the USBFS component.

Endpoint Descriptor

Endpoint descriptor specifies the characteristic of the endpoint such as the endpoint number (EP1 to EP8), direction (In, Out), transfer type and maximum packet size. **Endpoint** 0 (EP0) is used as a control endpoint and does not have a separate descriptor. It is always present in every design. The host will examine the endpoints in the device and determine the bus bandwidth at device enumeration.

The USBFS component provides access to endpoint descriptor fields in the **Device Descriptor** tab. You may instantiate multiple endpoint descriptor instances under an interface descriptor.

String Descriptor

String descriptors are optional descriptors that provide user readable information about the device to the host. These include strings such as the device name, manufacturer, serial number and names of interfaces/configurations. If a string is not used, it must be set to value 0x00.

The USBFS component allows string descriptor definition in each of the above mentioned descriptors, where applicable. These are collected and also shown in the **String Descriptor** tab. If a string field is left blank, the component will take care of setting its value to 0x00.

Miscellaneous Descriptors

Aside from the above 5 common descriptor types, the USBFS component provides access to define the following optional descriptor types.

- **Report Descriptors**: Used for providing extended descriptor information and attributes. The report descriptor is needed for a HID class. This is accessed in the **HID Descriptor** tab in the USBFS component.
- **MS OS Descriptor**: Provides Microsoft Windows OS with Windows specific information. This is optional, and is configured in the **String Descriptor** tab in the USBFS component.
- **BOS Descriptor**: Used to support Link Power Management (LPM) in PSoC 4200L devices. Using this improves idle power consumption by speeding up the time to enter and exit low power mode. You may use it by adding a BOS descriptor in the **Device Descriptor** tab in the USBFS component.

Refer to the Component Parameters section for more detail on these descriptor types.

Input/Output Connections

This section describes the various input and output terminals for the USBFS component. An asterisk (*) in the list of terminals indicates that the terminal may be hidden on the symbol under the conditions listed in the description of that terminal.

Dp – In/Out*

This terminal is the Data plus input of the USB Component. This terminal is always hidden. This input terminal is listed in the **Pins** tab of the *<project>.cydwr* file.

Dm – In/Out*

This terminal is the Data minus input of the USB Component. This terminal is always hidden. This input terminal is listed in the **Pins** tab of the *<project>.cydwr* file.

sof – Output *

The start-of-frame (sof) output allows the device 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 is selected.

vbusdet – Input *

The vbusdet input provides the ability to connect the host VBUS for voltage monitoring.

- This input is visible if the **VBUS Monitoring** and **IO pin external to the component** parameters in the **Advanced** tab are selected.
- This input is hidden on the symbol and VBUS pin is available only in the *<project>.cydwr* file if the **VBUS Monitoring** and **IO pin internal to the component** parameters in the **Advanced** tab are selected.

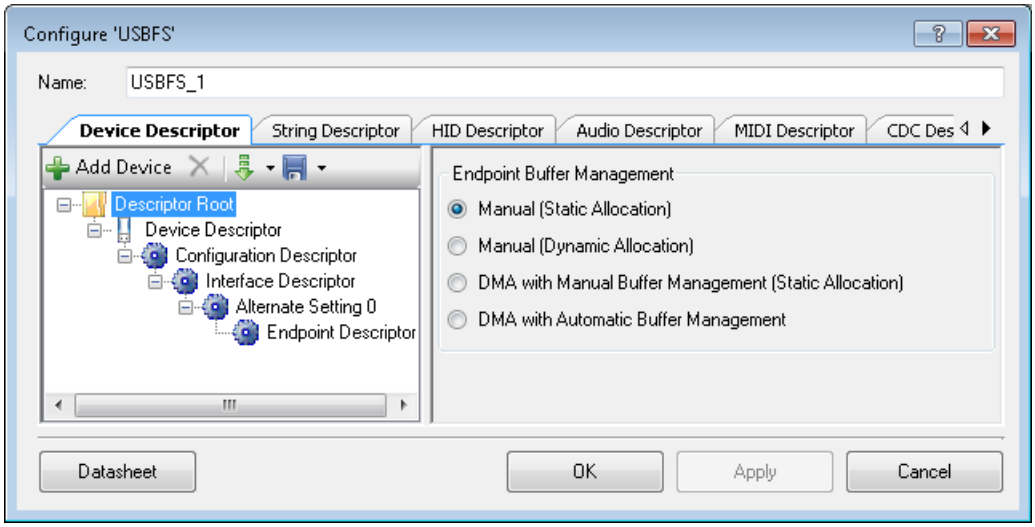
Component Parameters

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.

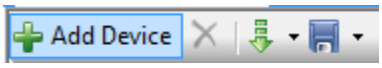
To begin, drag a USBFS component onto your design and double-click it to open the Configure USBFS dialog. The Configure USBFS dialog contains the following tabs and settings:

Device Descriptor Tab



This tab is used to configure Device Descriptors. It contains a menu and a Descriptor Root tree to select and configure different aspects of the Device Descriptors.

Menu items



Add <Level> Button

The **Add <Level>** button allows you to add a specified item. The added item depends on the level of hierarchical in the Descriptor Root currently selected. The items include:

- Descriptor Root level: the Device Descriptor can be added.
- Device Descriptor level: the Configuration Descriptor or Binary Device Object Store (BOS) Descriptor can be added.
- Configuration Descriptor level: the Interface Descriptor can be added. The drop down list provides following choices:
  - General, Audio
  - MIDI, CDC
  - MSC or Association

  Note Audio, MIDI, CDC, MSC interface descriptors must be created on the appropriate tab before they appear in drop-down menu.
- Interface Descriptor level: the Alternate Settings can be added.
- Alternate Settings level: the Endpoint Descriptor can be added.

Delete Button

The **Delete** button allows you to remove the selected item in the Descriptor Root.

Import Button

The **Import** button allows you to import a descriptor configuration. In the drop-down list, you can choose either:

- **Import Current Descriptor** – Loads the configuration of the selected descriptor.
- **Import Root Descriptor** – 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 import and save descriptor configurations that are configured on those tabs.

The USBFS Component also provides a set of pre-defined descriptor templates for commonly used configurations. These include the descriptors for HID class devices such as 3-button mouse, 5-button joystick, Audio class descriptors, generic HID data transfers and MSC class descriptor. You can import existing descriptors using the option **Import Descriptor Root (Ctrl+O)**. All the descriptor templates are provided in the following location:

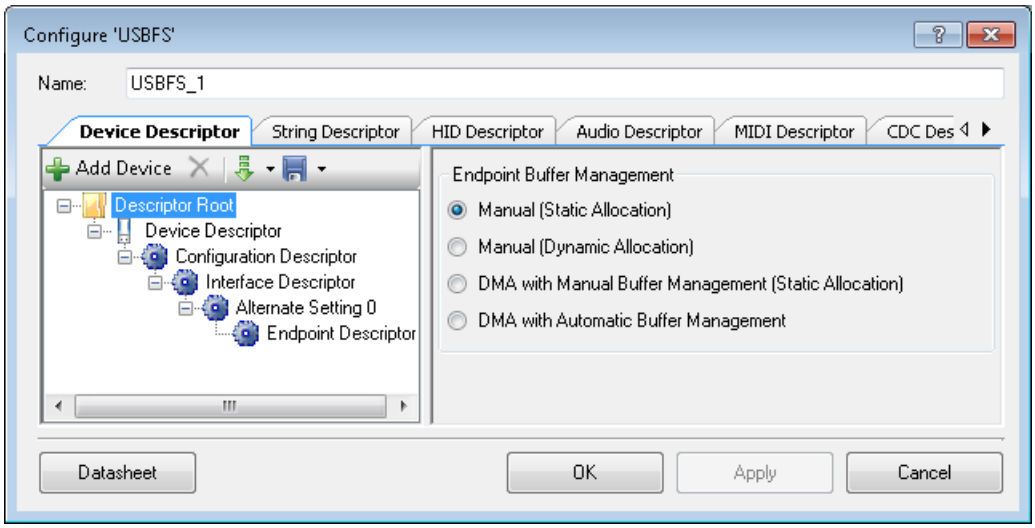
<PSoC Creator Installation Folder>
\psoc\content\cycomponentlibrary\CyComponentLibrary.cylib\USBFS_v3.0
Custom\template

Save Button

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** – Saves the configuration of the selected descriptor.
- **Save Root Descriptor** – Saves the whole device descriptor tree.

Descriptor Root



When you select the Descriptor Root level of the tree, the Endpoint Buffer Management section allows you to select the appropriate parameter.

Endpoint Buffer Management

The USBFS block has a 512 bytes internal buffer. This buffer is used to temporarily store the data transferred from and to the host through the endpoints. Each endpoint (which is available in device descriptors) has a space allocated in the buffer that belongs to it. The following options are how the endpoint buffer can be allocated and managed.

- **Manual** (default) – The CPU transfers data between system SRAM and the endpoint buffer manually. The functions USBFS_LoadInEP() or USBFS_ReadOutEP() must be called to execute data transfer. These functions return when the CPU completes data transfer and the endpoint is released to the host to be read or written.

  Manual endpoint buffer management provides two options for the endpoint buffer allocation as follows:

  - **Static Allocation** – The buffer size is fixed for each endpoint and allocated statically based on the device descriptor. The endpoint buffer size is equal to the maximum packet size across all endpoints of the same number within the device configuration. The buffers for the endpoints are allocated immediately after a SET_CONFIGURATION request is received. This option is preferred when the endpoint maximum packet size is the same or similar between alternate settings under the same interface or between configurations.

    For example, a device with a single configuration and interface has two alternate settings where endpoint 1 maximum packet size is 8 (alternate settings 0) and endpoint 1 maximum packet size is 512 (alternate settings 1). The buffer size allocated for endpoint 1 is 512 bytes. The endpoint 1 buffer consumes all available buffer space and it is not possible to add more endpoints to alternate settings 0. To overcome that, the dynamic buffer allocation must be used in this case.

  - **Dynamic Allocation** – The buffer size for each endpoint is dynamically allocated based on the current alternate settings. The buffer allocation occurs when a SET_CONFIGURATION or SET_INTERFACE request is received. This option is preferred when the endpoint maximum packet size differs between alternate settings under the same interface or between configurations.

- **DMA with Manual Buffer Management** – The DMA transfers data between system SRAM and endpoint buffer manually. The functions USBFS_LoadInEP() or USBFS_ReadOutEP() must be called to initiate DMA data transfer. These functions return when the DMA is initialized and data transfer has been started. How the DMA transfer completion event is handled depends on endpoint direction:

  - IN direction: call the USBFS_LoadInEP() function to initiate DMA data transfer from system SRAM into the endpoint buffer. After data has been transferred, the component is notified by the interrupt and releases the endpoint to be read by the host in the next IN transaction. The endpoint state is changed from buffer empty to full when the DMA transfer has been completed.
  - OUT direction: call the USBFS_ReadOutEP() when data is in the endpoint buffer (endpoint state is buffer full) to initiate DMA data transfer from the endpoint buffer

into system SRAM. After data has been transferred, the component is notified by the interrupt and changes endpoint state from buffer full to empty. To allow host writing new data into the OUT endpoint the USBFS_EnableOutEP() has to be called when endpoint buffer is empty.

Only static buffer allocation is allowed for this endpoint buffer management mode.

- **DMA with Automatic Buffer Management** – The DMA transfers data between system SRAM and endpoint buffer automatically. The functions USBFS_LoadInEP() or USBFS_ReadOutEP() must be called once to initialize the DMA and register system SRAM as an extension of the endpoint buffer. The DMA automatically accesses the system SRAM while communicating via USB. This approach provides the ability to get the total endpoint buffer, which exceeds 512 bytes.

  The automatic buffer management implies the special endpoint buffer allocation scheme. The 512 bytes USBFS buffer is divided between endpoints as follows:

  - 32 bytes are allocated for each active endpoint (endpoint which presents in the device descriptors).
  - Remaining space is left for the common area. The common area is used as FIFO to accept the data when endpoint buffer is full while communication.

  The USBFS block generates DMA requests as soon as communication starts to keep common area fully loaded for IN direction and keep common area empty for OUT direction.

  For example, device uses all 8 endpoints. The endpoint buffers consume 8 * 32 = 256 bytes, remaining bytes is common area 512 – 256 = 256 bytes. If only six endpoints are used, the common area size would then be 320 bytes (512 - (6 * 32)).

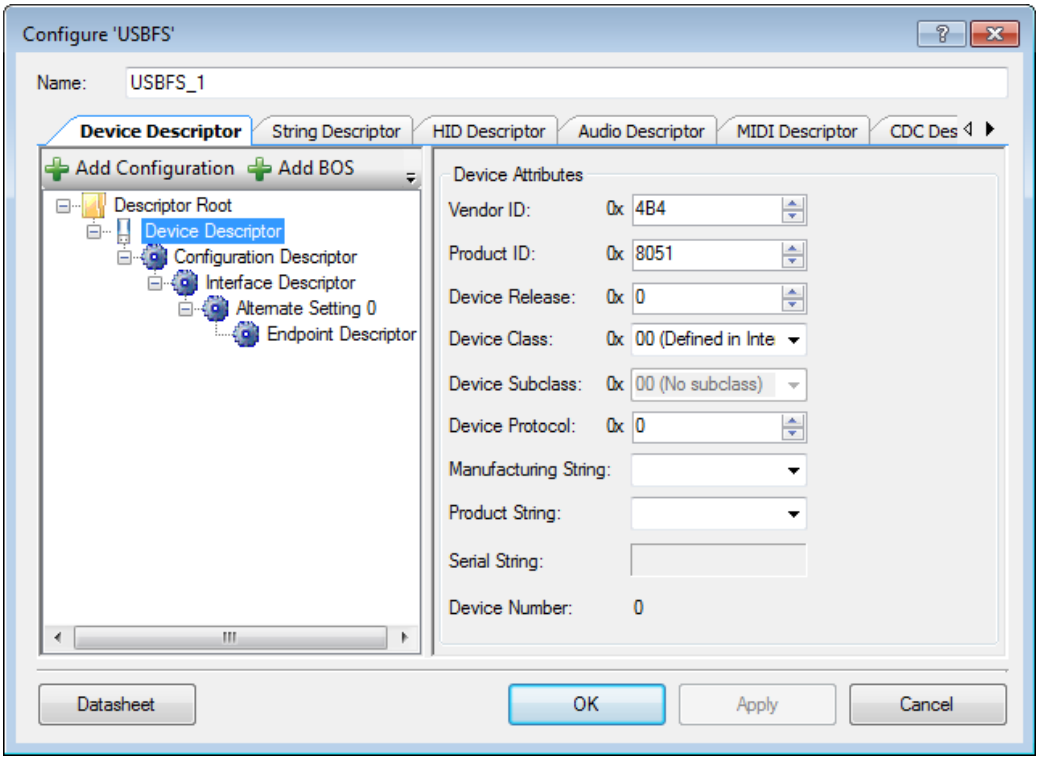
For more details about operation in any of modes described above, refer to the USBFS Basic Workflow In Different Modes section.

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

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

Device Descriptor

A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device can have only one active device descriptor. There should be at least one device descriptor present under the Descriptor Root; otherwise, the PSoC Creator will generate an error when the project is built.



Device Attributes

- **Vendor ID** – This is a 16-bit number used to uniquely identify **USB** devices belonging to a specific vendor/manufacturer to a USB host. Vendor IDs are assigned by the USB Implementers Forum (USB-IF). The following link in the USB-IF webpage explains the method to obtain a vendor ID for your company:

  http://www.usb.org/developers/vendor/

  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** – This is a 16-bit number assigned by the device vendors/manufacturers to uniquely identify a USB device to the USB host.

- **Device Release** – The device release is a 16-bit number and is used for versioning the device. For example, this field can be used to identify the version of the firmware running in the device and to determine if a firmware upgrade is required or not.

- **Device Class** – This parameter defines the class of the USB device and is communicated to the USB host at the time of device enumeration. It allows the host recognize the device functionality and load the appropriate device driver to enable that functionality. Some device classes can be defined in the **Device Descriptor**, whereas others are defined in the **Interface Descriptors**. Refer to the official USB Class Codes page for more info.

  If this field is set to **Defined in Interface Description** in the GUI, each interface specifies its own class information and operates independently. If any other value is selected from the drop-down menu (or manually entered), the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. Device class options include number assigned in specification and description. The following options are available in the drop down menu:

  - **00 (Defined in Interface Descriptor)** – Interface specific device classes
  - **02 (CDC)** – Communications and CDC Control defined in the Device Descriptor
  - **FF (Vendor-Specific)** – Vendor specific class defined in the Device Descriptor

  Enter hexadecimal number manually if the drop-down menu does not provide required device class.

- **Device Subclass** – This field is dependent on the value specified in the field **Device Class** and must conform to the subclass options as specified in the official USB Class Codes. If the **Device Class** field is reset to zero, this field must also be reset to zero. If the **Device Class** field is not set to Vendor Specific, all values are reserved for assignment by the USB-IF. The default value for this parameter is "**No subclass**", which is the case for many device classes.

- **Device Protocol** – This is an 8-bit number and is qualified by the value of the **Device Class** and the **Device Subclass** fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class specific protocols on an interface basis. If this field is set to 0xFF, the device uses a vendor-specific protocol on a device basis.

- **Manufacturing String** – This is an optional field and the string can be specified by the manufacturer. The manufacturer description string is displayed when the device is attached. The maximum length of the manufacturing string is 126 characters.

- **Product String** – Defines the product-specific description string to be displayed when the device is attached.

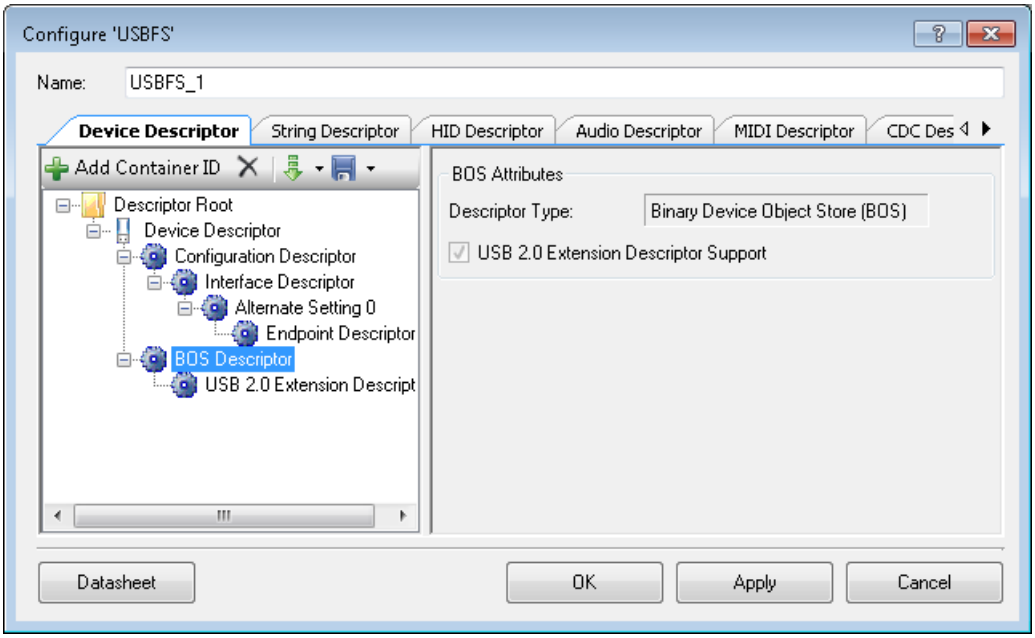
- **Serial String** – This is an optional read-only field specified by the manufacturer in the String Descriptor Tab or PSoC using USBFS_SerialNumString(). PSoC generates a serial number string based on the die ID of the PSoC device. This field is used to distinctly identify USB devices if multiple devices with same VID and PID are attached to the same

host. The maximum length of the serial string is 126 characters. Also refer to **Serial Number String** parameter in the **String Descriptor** tab.

- **Device Number** – This is read-only field, which identifies device descriptor in descriptor tree. The USBFS Component allows you to create more than one USB device descriptor setting. Click on “Descriptor Root” in the tree view and click on **Add Device** to add another device. Each setting is numbered automatically by the GUI uniquely starting from 0. This number is used to identify which descriptor is sent to the host when PSoC device tries to enumerate. The selection is performed using the function USBFS_Start(), which accepts the device number and desired mode of operation as its parameters.

BOS Descriptor

USBFS component supports Link Power Management (LPM) mode for PSoC 4200L devices. This feature allows improved power management and is especially useful when we need to put the device in Idle/Suspend state for a short period of time (for example, between communication to save energy for mobile devices). The BOS Descriptor allows the device to include the LPM capability. Refer to the LPM specification for more information.



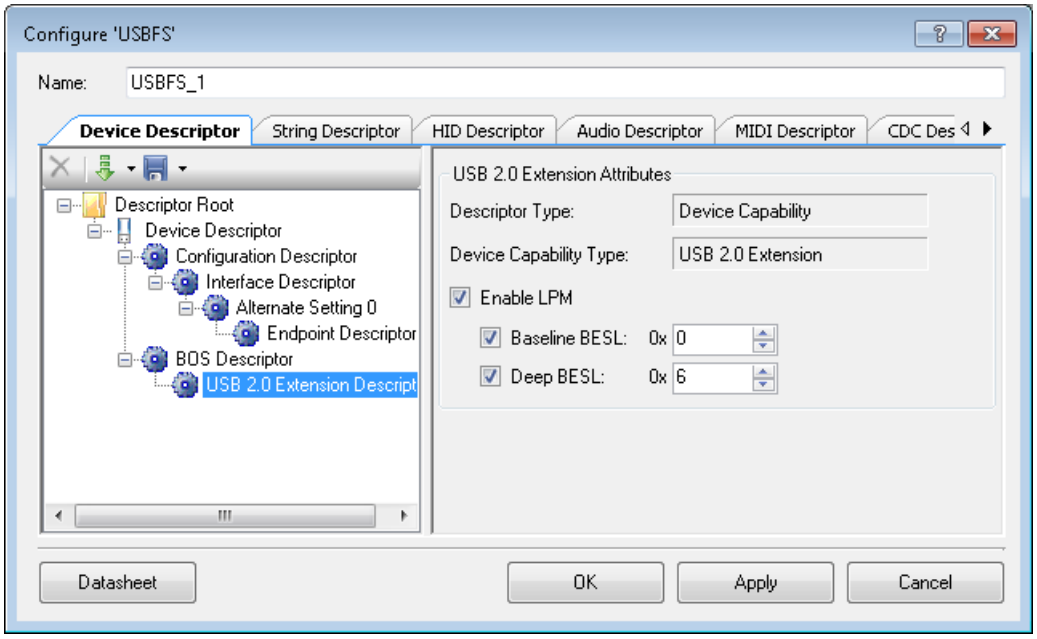
Note If a BOS descriptor is added, the device may trigger a message on the host that says the USBFS device could transfer faster if connected to the Superspeed USB 3.0 port. For example, Windows OS triggers this message based on the bcdUSB field value of the Standard Device Descriptor when it is greater than 2.0. The USB component sets the bcdUSB value to 2.01 according to the LPM specification, when the BOS descriptor is supported. The communication will be limited to USB 2.0 speed regardless of this message.

BOS Descriptor Attributes

- **Descriptor Type** – Specifies that this is a BOS descriptor.

- **USB2.0 Extension Descriptor Support** – This is a read-only field which specifies that the USB 2.0 Extension descriptor is supported.

USB2.0 Extension Descriptor



USB2.0 Extension Descriptor Attributes

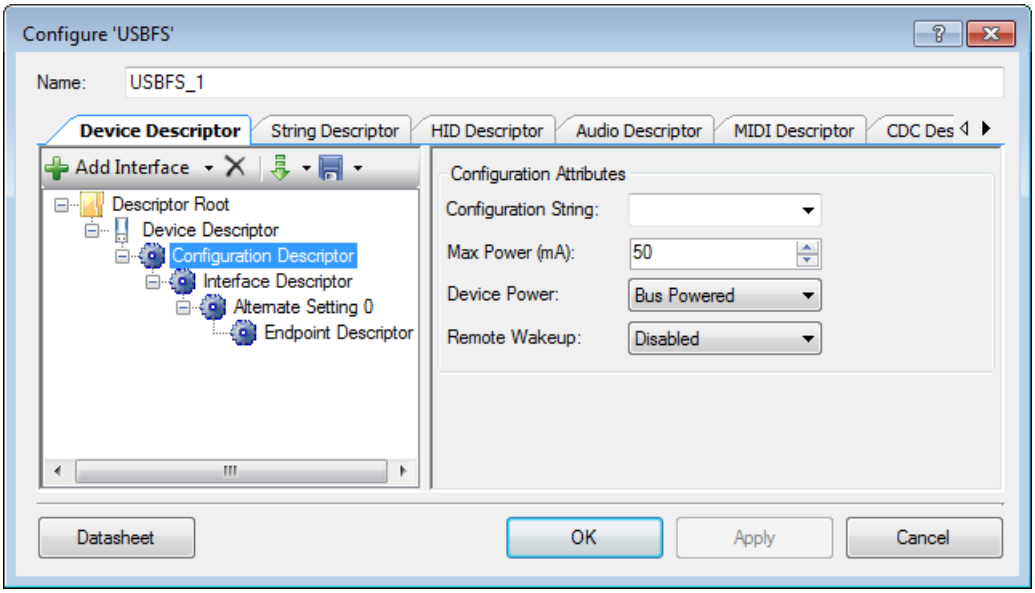
- **Descriptor Type –** This is a read-only field that specifies the type as a “Device Capability”.
- **Device Capability –**This is a read-only field that specifies the capability type as a “USB 2.0 Extension”.
- **Enable LPM –** This option enables the LPM support for the device which will be reported to the host during device enumeration. Enabled by default.
  - **Baseline BESL** – This check box enables the baseline Best Effort Service Latency (BESL) value. When it is checked, you can specify the expected latency value in the given field. This value is the expected latency from the beginning of a resume signaling to the start of transactions to the device. This value will be sent to the host during enumeration. Default value is 0.
  - **Deep BESL** – This check box enables the deep BESL value. When it is checked, you can specify the expected latency value that can be greater than the baseline BESL in the given field. This value will be sent to the host during enumeration. Default value is 6. Recommended value is 8 or 9.

  It is recommended that the baseline BESL value should be less than deep BESL. The expected use case is that the baseline BESL value communicates a nominal power

savings in a design, whereas the deep BESL value communicates a significant power savings in a design. The PSoC 4200L device provides a choice of low power modes to select the desired power saving strategy. Refer to the *PSoC Creator System Reference Guide* for more information about power management. Note that the USBFS component provides only APIs to support Deep Sleep power mode for PSoC 4200L.

Configuration Descriptor

The configuration descriptor describes information about a specific device configuration. The descriptor describes the different interfaces provided by the configuration and each interface may operate independently.



Configuration Attributes

- **Configuration string** – This field is used to specify the string that is used to identify the configuration.
- **Max Power (mA)** – This number specifies the maximum possible current draw (power consumption) of the USB device from the bus when the device is fully operational. This is a Configuration Descriptor specific configuration. A low-powered device draws at most 100mA and a high-powered device draws at most 500 mA. The USB component supports both and hence the maximum value of this parameter is 500. The default value is 50 mA

  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** – Specifies whether it's a **Bus-powered** or **Self-powered** device. The USBFS component does not support both settings simultaneously. The default value is Bus-powered.
- **Remote Wakeup** – This option is used to specify if the USB device can wake-up the host from a low power state. If this option is **Enabled**, the host wakes up on a signal from the USB device. Before that the host must give device permission to execute remote wake-up. This parameter is **Disabled** by default.

Interface Association Descriptor

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

If a logical device function is performed using more than one interface in a USB device, the host should be notified of this fact so that the host uses only a single driver to address this function. Otherwise the host will recognize each interface as a separate logical function and may fail to communicate with the USB device.

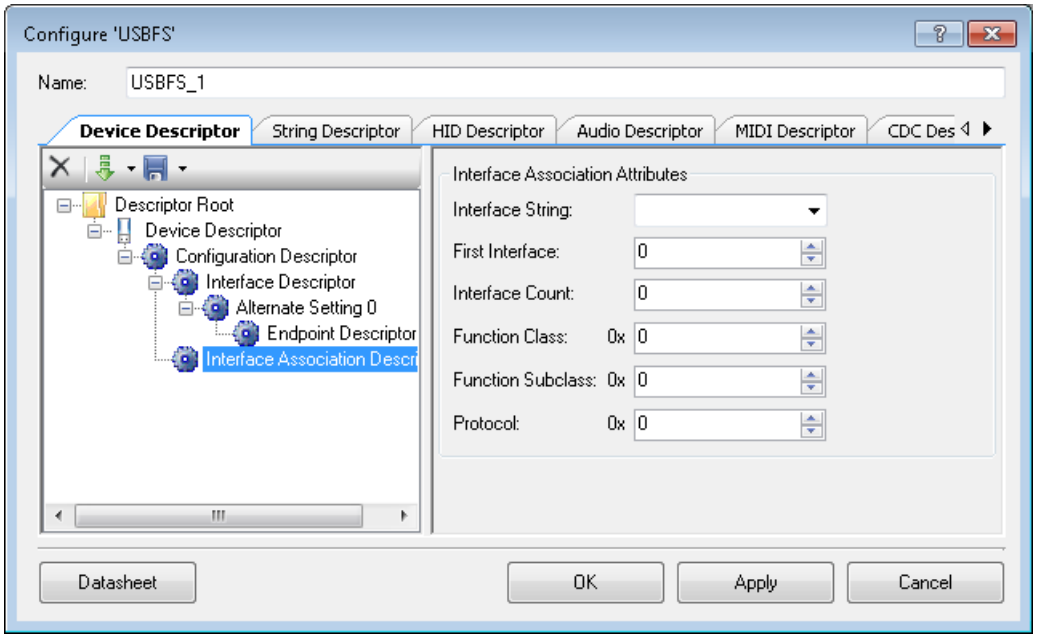
Note The IAD must be positioned just above the interfaces that need to be grouped together and only contiguously numbered interfaces can be associated using the IAD.

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**.

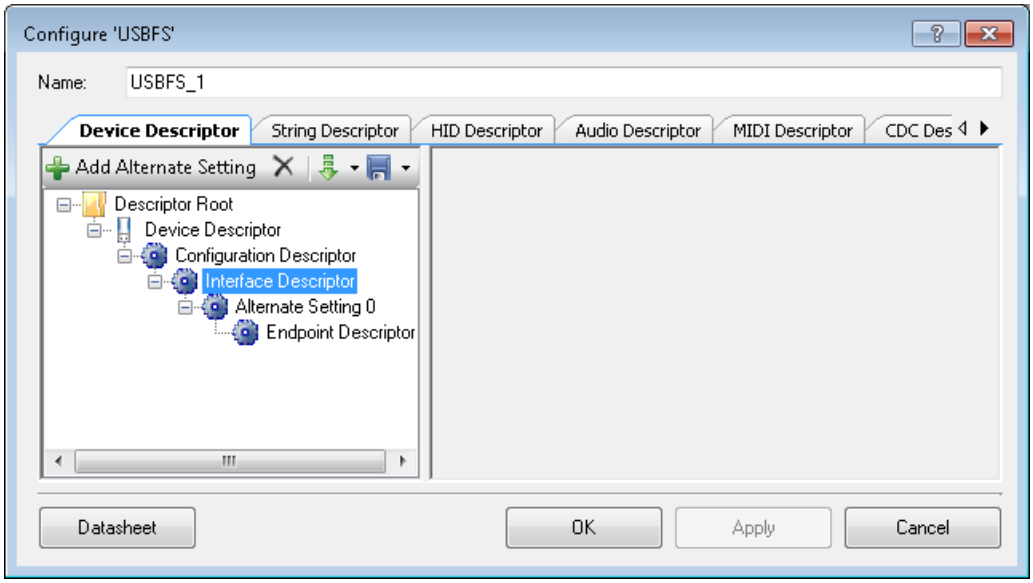


Interface Association Attributes

- **Interface String** – This is an optional field. This field specifies the string that should be used to identify the Interface Association Descriptor when the device is connected to the host.
- **First Interface** – Interface number of the first interface associated with this function. The interface number can be found in the alternate settings of the corresponding interface beside the label **Interface Number**.
- **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** – This field has the same value as subclass code in the first associated interface.
- **Protocol** – This field has the same value as protocol code in the first associated interface.

Interface Descriptor

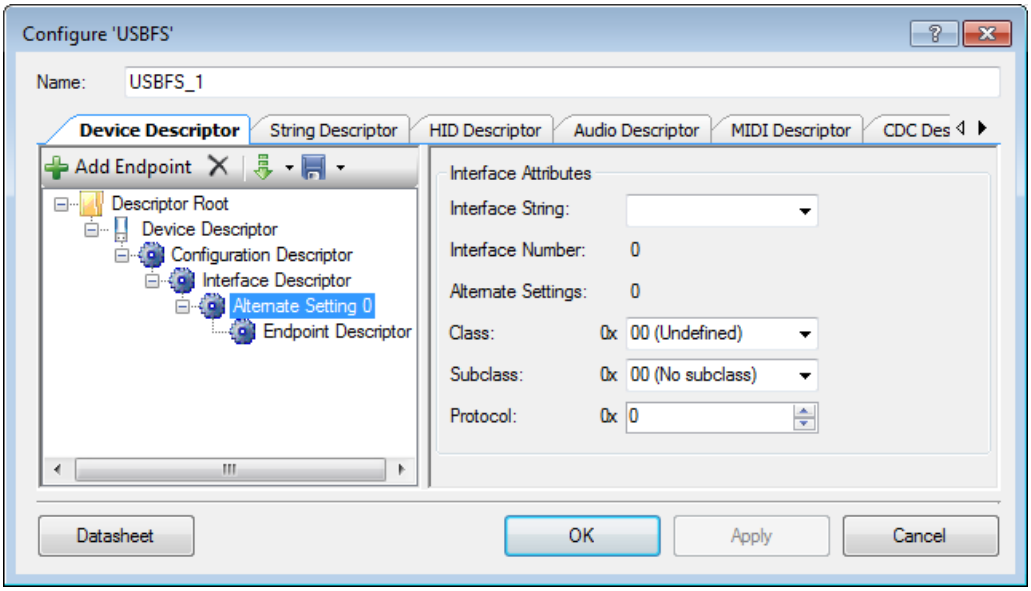
The interface descriptor describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always **Alternate setting 0**. Each device descriptor must contain at least one interface descriptor.



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 by adding an Alternate Setting in your tree view. 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.

Alternate Settings



Interface Attributes

- **Interface String** – This is an optional field. The string specified in this field is displayed as the interface string in the host to identify the corresponding interface.
- **Interface Number** – This is a read-only value used to identify the index of the interface in the array of concurrent interfaces supported by this configuration. This value is computed by the GUI for you since the interface numbers in a configuration always starts with zero and is numbered consecutively for the successive interfaces in a configuration.
- **Alternate Settings** – This is a read-only value used to identify this alternate setting for the interface identified in the Interface Number field. This value is computed by the GUI for you since the alternate setting numbers within an interface always starts with zero and is numbered consecutively for the successive alternate settings in an interface.
- **Class** – This field is used to identify the capabilities of the device and to load a device driver based on that functionality. The **Device Class** parameter in the **Device Descriptor** should be set to "**Defined in Interface**". The information is contained in three bytes with the names Base Class, Subclass, and Protocol. A list of class codes is available in the USB-IF webpage: http://www.usb.org/developers/defined_class.
  - **00 Undefined** – No class specified (Default)
  - **03 (HID)** – Human Interface Device base class
  - **FF (Vendor-Specific)** – Vendor specific class

  Enter hexadecimal number manually if the drop-down menu does not provide required Class. The Interface Class code may also be manually entered in the field.

- **Subclass** – Dependent on the selected class. Default is **No subclass**.
- **Protocol –** Dependent on the selected class and subclass. Default is **0x0**.

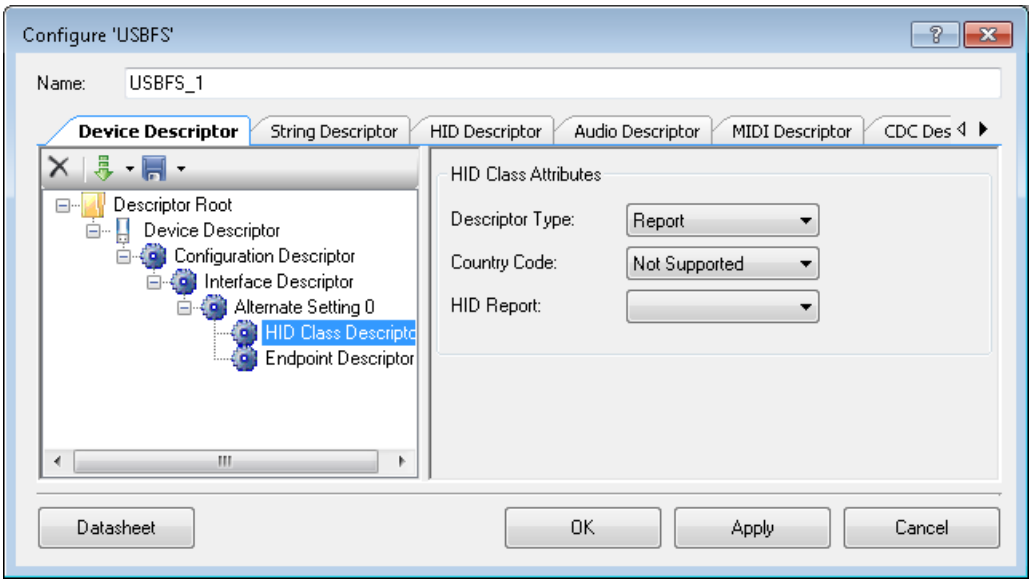
Note String descriptors are optional. If a device does not support string descriptors, all references to string descriptors within the device, configuration, and interface descriptors must be set to zero.

HID Class Descriptor

The HID Class Descriptor is added when the **Interface Descriptor**'s **Class** parameter is set to "HID".

To Add HID Class Descriptor

1. Select an **Alternate Setting** item in the **Descriptor Root** tree.
2. Under **Interface Attributes** on the right, select **(03) HID** for the **Class** field.



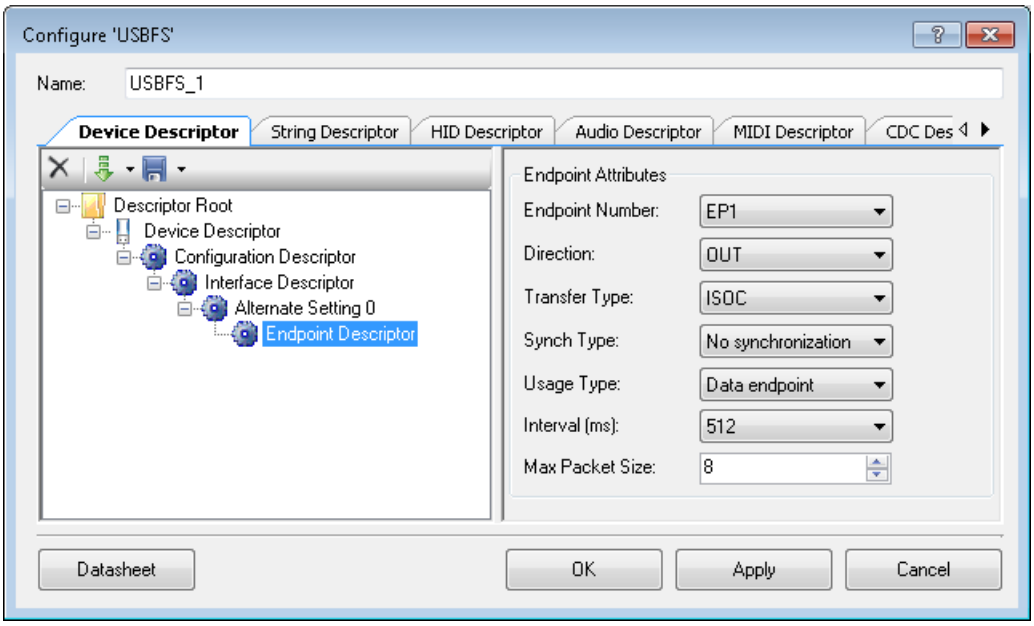
HID Class Attributes

- **Descriptor Type** – Constant name identifying the type of class descriptor.
  - **Report** – Define items that describe a position or button state.
  - **Physical** – Provide information about the specific part or parts of the human body that are activating a control or controls. For example, a Physical descriptor might indicate that the right hand thumb is used to activate button 5. An application can use this information to assign functionality to the controls of a device.
- **Country Code** – Numeric expression identifying the country code of the localized hardware. This is **optional** and, is provided to inform the host about the language specific

functions associated with the device, such as the alternate language supported by a USB HID keyboard.

- **HID Report** – This drop-down lists the available report descriptors. The report descriptors are taken from the **HID Descriptor** tab.

Endpoint Descriptor



Endpoint Attributes

- **Endpoint Number** –Select one of the available eight endpoints (EP1 to EP8). The USBFS GUI automatically removes the endpoints that are already used from the drop down menu, since the selected endpoint should not be selected in any other endpoint descriptor under the same Alternate Setting or by a different interface within a configuration.
- **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
- **Synch Type –** This field is displayed only if the **Transfer Type** is set to **ISOC**. This information is required in order to determine how to connect isochronous endpoints. The available options are **No synchronization**, **Asynchronous**, **Synchronous**, and **Adaptive**. A detailed explanation on synchronization is available in section 5.6 of the USB Specification Revision 2.0.

- **Usage type –** This field is displayed only if the **Transfer Type** is set to **ISOC**. It is used to indicate the purpose of this endpoint.
  - **Data endpoint** – Normal data transfers.
  - **Feedback Endpoint** – Explicit feedback information for one or more data endpoints.
  - **Implicit Feedback Data Endpoint** – Data endpoint that also serves as an implicit feedback endpoint for one or more data endpoints**.**
- **Interval (ms) –** Polling interval specific to this endpoint. For isochronous endpoints the polling interval choice is restricted to: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 (ms). For control and interrupt endpoints the polling interval can be any value between 1 and 255. This field is not applicable for **BULK** endpoints.
- **Max Packet Size –** (In bytes) For a full-speed device the **Max Packet Size** is 64 bytes for bulk or interrupt endpoints and 512 (1023 for DMA with Automatic Buffer Management mode) bytes for isochronous endpoints. For full-speed device bulk endpoints only 8-, 16-, 32-, and 64-byte values are allowed.

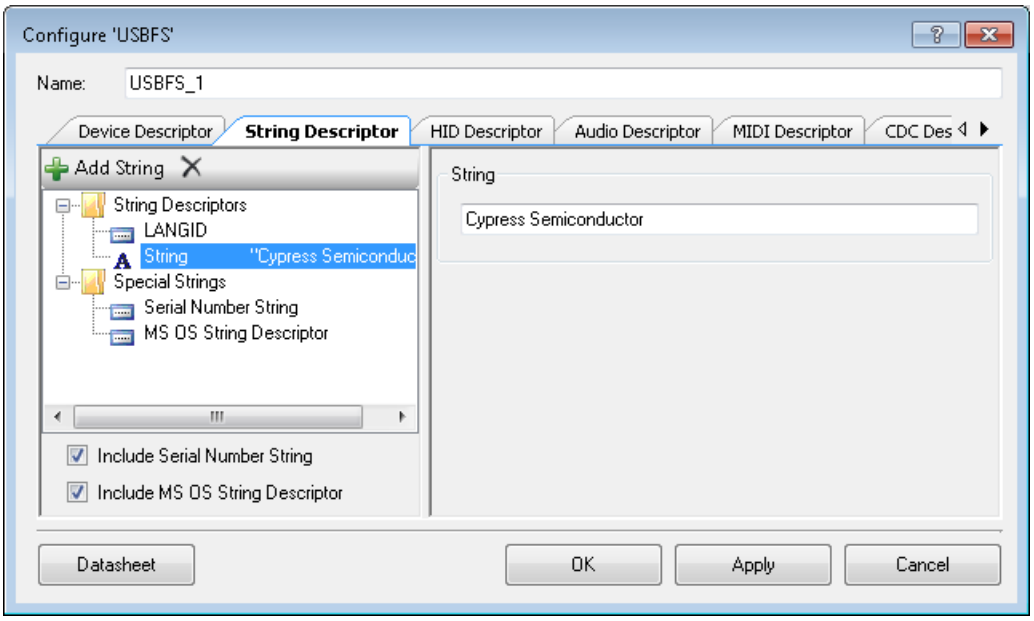
  The maximum packet size for the isochronous endpoints is limited by the USB block memory size, which is equal to 512 bytes, in the Manual Memory Management mode whereas in the DMA with Automatic Buffer Management DMA mode there is no has no such limitation because the USB block memory is treated as a temporary buffer.

String Descriptor Tab

This tab allows you to create and edit all the strings used in the Device Descriptor. The strings created in the **Device Descriptor** tab are automatically populated in the **String Descriptor** tab. You can also add new strings in the **string descriptor** tab and re-use these strings in different descriptors using the drop-down menu option in the corresponding descriptor configurations.

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 left blank or set to 0.

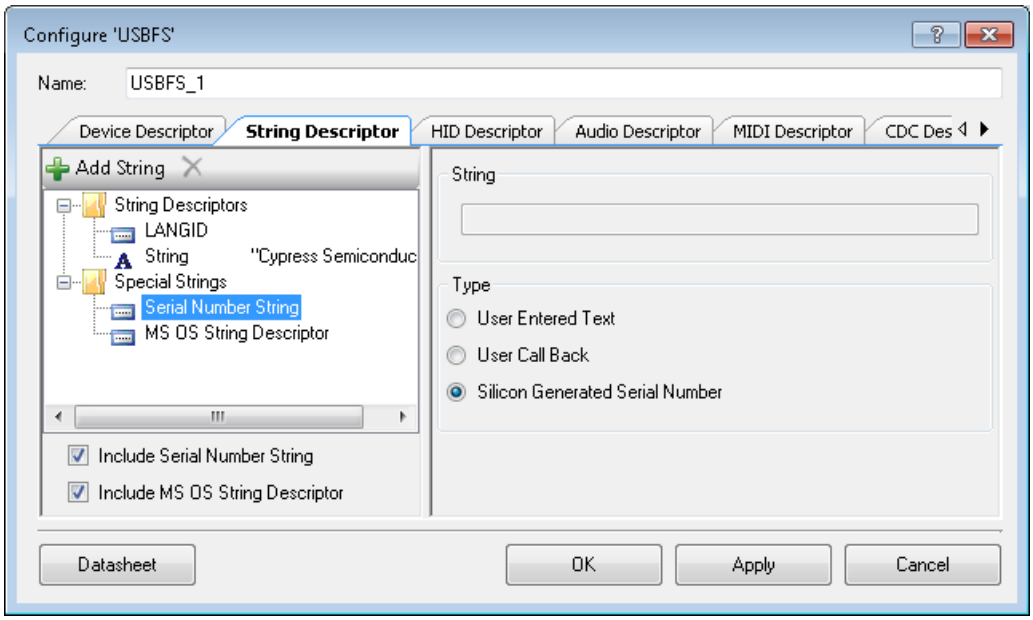
String Descriptors



- **LANGID** – This indicates which language (English, German, Chinese, etc.) to use to retrieve words in String Descriptors.
- **String** – The text used for string descriptor.

Include Serial Number String

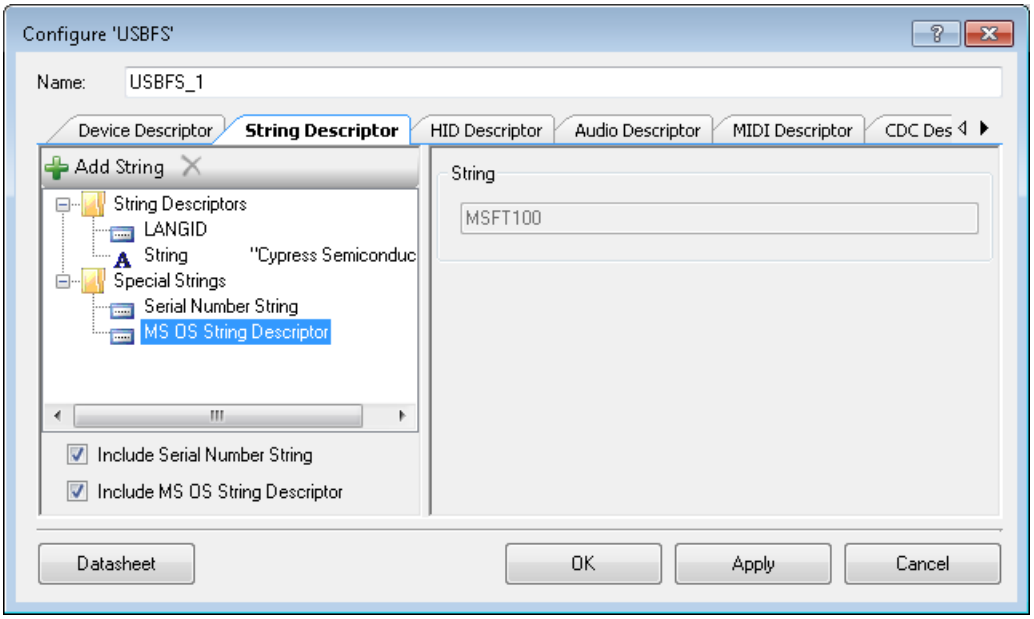
This parameter adds the serial number string that is used in the device descriptor. **Include Serial Number String** checkbox must be checked in order to expose this option.



- **String** – This field is used to enter the string that should be used for the string descriptor. It can only be entered if the **Type** is set to **User Entered Text**.
- **Type** – One of three options to specify the String, as follows:
  - **User Entered Text** – You can specify the string to be used for the String Descriptor at design time. This is entered in the **String** field. If more than one device connected to the same host uses the same Vendor ID, Product ID, and Serial Number String, the USB host may not recognize all the devices. The USBFS custom dialog will display a warning to indicate this, but will not cause any build or compile errors.
  - **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 generated from the die ID of the silicon. The die ID is applied to non-volatile memory in the device at manufacturing time and it is not guaranteed to be unique.

Include MS OS String Descriptor

This option adds the MS OS String Descriptor that provides a way for USB devices to supply additional configuration information to the Microsoft operating systems, beginning with Windows XP and above. Microsoft operating systems allow USB devices to be used by a Windows application without having to install device drivers even if the USB device does not belong to HID or Mass Storage Class. This requires the device descriptor to have a string descriptor with string “MSFT100” at a specific location in the device descriptor.

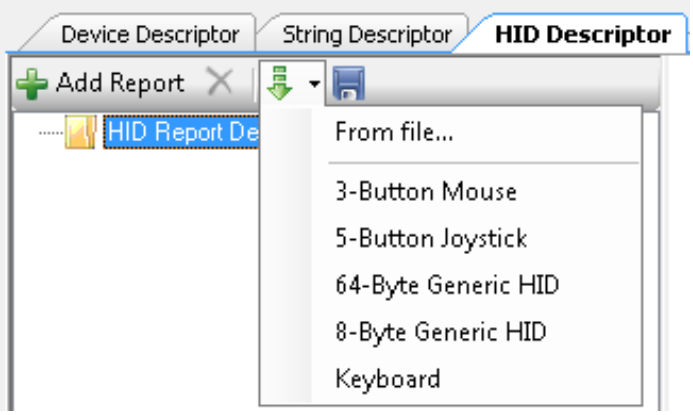


- **String** – Constant string “MSFT100”.

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.

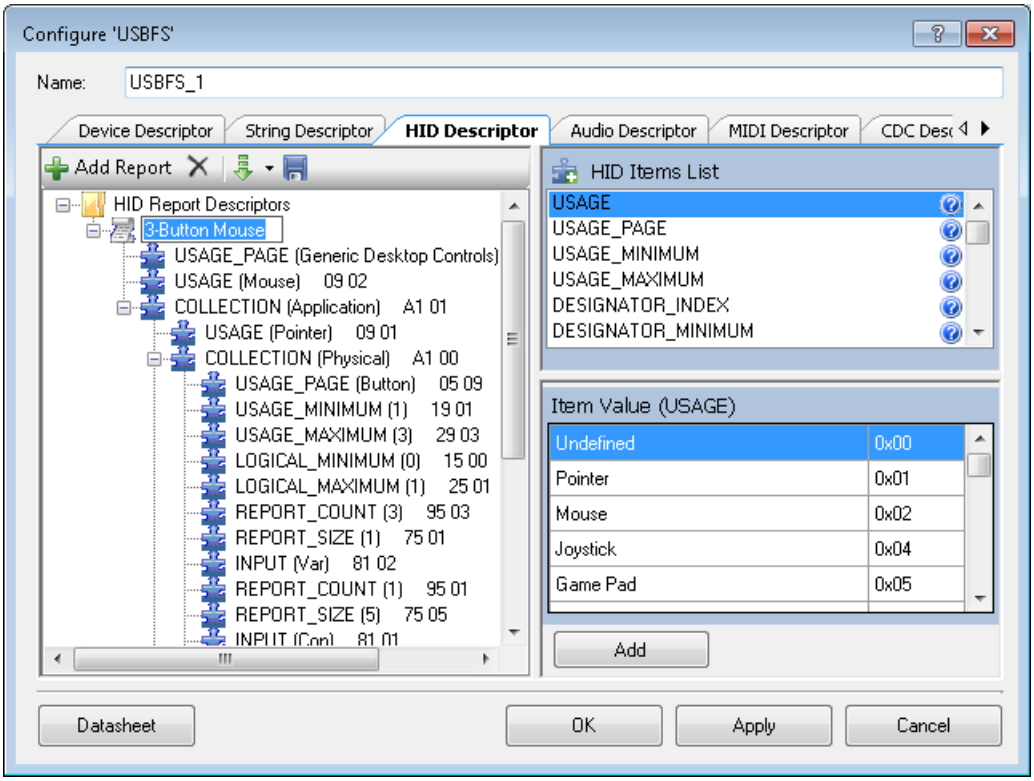
Use the **Import** button to import the HID report. In the drop-down list you can choose one of the templates or **From file**.

The template options immediately load the selected HID report. The **From file** option will open a 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** – Report descriptors are composed of pieces of information. Each piece of information is called an Item. This area allows you to add Items to add in the HID report. Detailed information on the HID descriptors is available in the Device Class Definition for Human Interface Devices (HID).
- **Item Value (USAGE)** – This area allows you select a value that is appropriate for the currently selected HID item. The parameters in these windows are context based and will vary depending upon the item value selected in the HID Items List window.

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

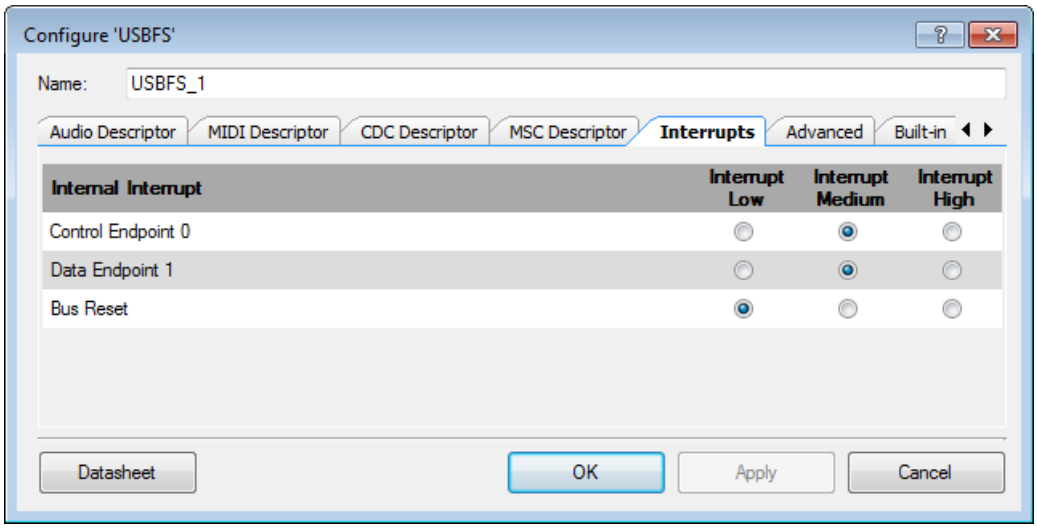
The **CDC Descriptor** tab is used to add and configure communications and data interface descriptors. See the USBUART (CDC) section for more information.

MSC Descriptor Tab

The **MSC Descriptor** tab is used to add and configure Mass Storage Class interface descriptors. See the USBFS MSC section for more information.

Interrupts Tab

The Interrupts tab is only visible for PSoC 4200L devices and used to map interrupt sources to the hardware interrupts.



The USB block has 13 interrupt sources which are mapped to the 3 hardware interrupts. These interrupts are named Interrupt Low, Interrupt Medium, and Interrupt High. The USBFS component automatically maps interrupt sources to an interrupt and this can be used as a guideline to assign the actual interrupt priority in the *<project>.cydwr* file. You can change the interrupt to which each interrupt source it belongs to by using the radio buttons. The interrupt sources are visible only when they are used for the USBFS component operation. Refer also to the Interrupt Service Routine section to get more details about interrupts utilized by USBFS component.

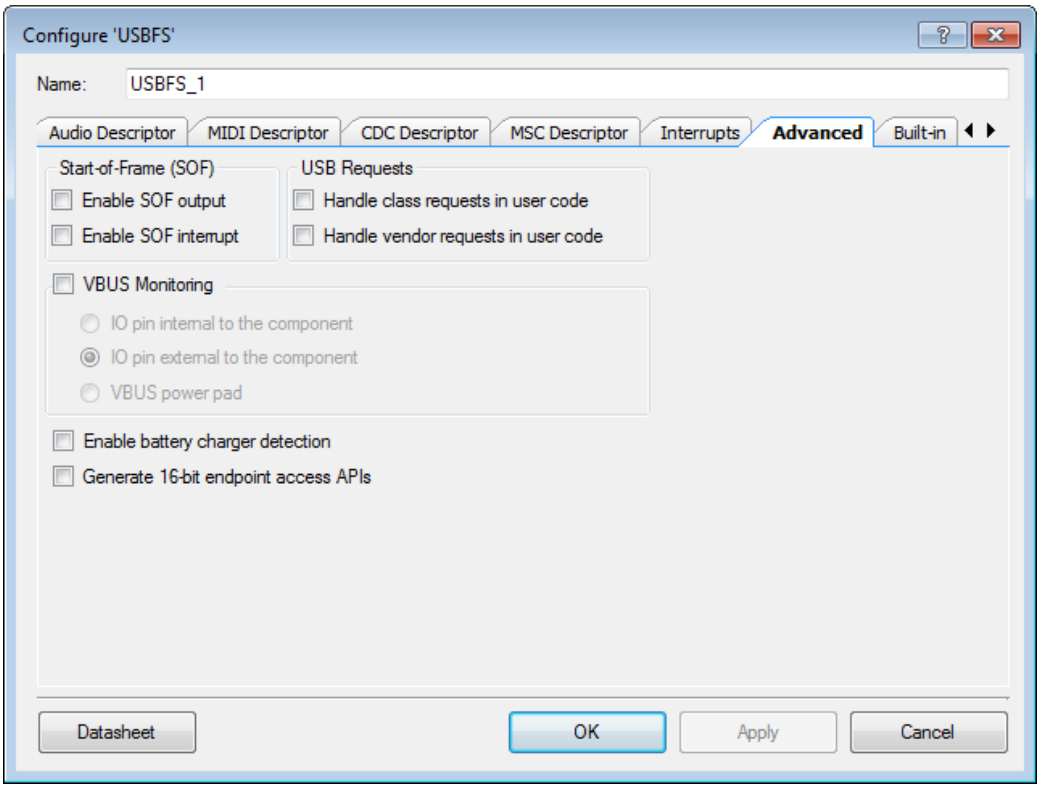
Note This tab is used only to map all the available interrupt sources to three interrupts. The actual priority level for an interrupt is assigned in the **Interrupts** tab of the *<project>.cydwr* file.

- **Start of Frame (SOF)** Interrupt – This interrupt source triggers when a start of frame is received. To enable this interrupt source, the Enable SOF interrupt option has to be checked.
- **Bus Reset** Interrupt – This interrupt source triggers when a USB bus reset even occurs. It is mandatory for USBFS component operation.

- **Control Endpoint 0** Interrupt – This interrupt source triggers whenever the host tries to communicate over the control endpoint. It is mandatory for USBFS component operation.
- **Data Endpoint 1-8** Interrupts – These interrupt sources trigger whenever the host completes communication over the corresponding data endpoint. They are available only when an endpoint is utilized by the device.
- **Link Power Management (LPM)** Interrupt – This interrupt source triggers when an LPM entry USB extension packet is received. It is available when there is at least one BOS descriptor present in the Device Descriptor tab.
- **Arbiter Interrupt –** This interrupt source triggers when one of the following conditions for any endpoint are met: IN endpoint buffer full, endpoint DMA grant, endpoint buffer overflow, endpoint buffer underflow, endpoint error in transaction, endpoint DMA terminated (applicable for PSoC 4200L). It is available only when Endpoint Buffer Management is set to **DMA with Manual Buffer Management** or **DMA with Automatic Buffer Management**. Refer to the Interrupt Service Routine section.

Advanced Tab

This tab is used to provide advanced options related to vendor specific firmware implementations and hardware level configurations.



Enable SOF output

This parameter exposes the sof output on the USBFS symbol. The Start-of-Frame (SOF) output allows the device to identify the start of the frame and synchronize application events with it. The host issues SOF packets at a nominal rate of once every 1 ms for a full-speed bus. The host stops sending SOF packets for more than 3 ms to suspend the USB device. This option is unchecked by default.

Enable SOF interrupt

This parameter enables the interrupt generation when a SOF packet is received from the host. The component provides empty SOF interrupt handler which takes care about clearing interrupt source. The user code can be inserted inside this interrupt handler use callback macros. Refer to the Macro Callbacks section for more details. This option is unchecked by default.

Handle class requests in user code

The component provides class requests handler for the supported classes (available in the *USBFS_cls.c* file). This parameter overrides the class request handler function with a user implementation. The USBFS_DispatchClassRqst() function must be implemented by the user to service incoming class requests. This function is invoked whenever the class request is received and must return with an indication on whether it was handled or not. This option is unchecked by default.

Handle vendor requests in user code

The component provides an empty vendor request handler (available in the *USBFS_vnd.c* file) for unsupported vendor requests. This parameter overrides the vendor request handler function with a user implementation. The USBFS_HandleVendorRqst() function must be implemented by the user to service incoming vendor requests. This function is invoked whenever the vendor request is received and must return with an indication on whether it was handled or not. This option is unchecked by default.

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 Dp/Dm pull-up resistor if VBUS is absent.

Note For bus-powered designs, power is removed when the USB cable is removed from the host. It is imperative that for proper operation and USB certification, your device complies with this requirement.

Note For self-powered designs, refer to the USB Compliance for Self-Powered Devices section.

By default, **Enable VBUS Monitoring** checkbox is unchecked. There are three sub-options provided for VBUS monitoring:

- **IO pin internal to the component –** the VBUS pin is added inside the USBFS component. The input state of this IO is returned by USBFS_VBusPresent() function. The polling method should be used to monitor VBUS. The pin drive mode is High Impedance Digital but could be changed using the pin-specific function, USBFS_VBUS_SetDriveMode().

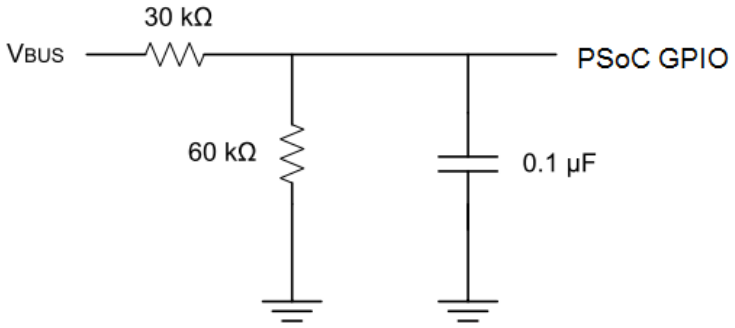
  Note For PSoC 4200L device VBUS pin must be assigned to P0[0].

- **IO pin external to the component** – vbusdet input terminal is exposed by the USBFS component. A digital input Pin Component must be connected to vbusdet terminal. General use case for this option is to detect the VBUS using the port specific interrupt logic on the IO pin rather than periodically reading the pin state. The input state of this IO is returned by USBFS_VBusPresent() function. This option is set by default if VBUS monitoring is enabled.

  Note For PSoC 4200L device pin connected to vbsudet terminal must be assigned to P0[0].

- **VBUS power pad –** This option is only applicable for **PSoC 4200L** devices. The USBFS component uses the VBUS power pad to return the VBUS status through the USBFS_VBusPresent() function. This option allows you to save GPIO pins using a dedicated power pin instead.

It is recommended to connect the VBUS through the resistive network, when the **IO pin external to the component** option is selected. Main aim of such connection is to save pin from voltage picks on VBUS. An example schematic is shown in the following figure.



For a PSoC 3/PSoC 5LP device, the VBUS monitoring pin can be directly connected to VBUS, if it is assigned to an SIO port. This configuration utilizes the hot swap capabilities of these pins.

Enable battery charger detection

This option is only applicable for PSoC 4200L devices. It allows you to detect USB host port with charging capabilities by using the function USBFS_Bcd_DetectPortType(). If the host supports battery charging, the device is allowed to draw up to 1.5 A of current from the host. This can be used to charge a battery connected to the system. This option is unchecked by default.

Generate 16-bit endpoint access APIs

This option is only applicable for PSoC 4200L devices. It enables generation of functions that execute 16-bit access to the endpoint data registers:

- void USBFS_LoadInEP16(uint8 epNumber, const uint8 pData[], uint16 length)
- uint16 USBFS_ReadOutEP16(uint8 epNumber, uint8 pData[], uint16 length)

These functions provide faster processing of endpoints data but they require special SRAM buffer allocation. For more information, refer to 16-bit Endpoint Access API section. This option is unchecked by default.

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.
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.

Function	Description
USBFS_Stop_DMA()	Stops DMA channel associated with endpoint.
USBFS_LoadInEP()	Loads and enables the specified USBFS endpoint for an IN transfer.
USBFS_LoadInEP16()	Loads and enables the specified USBFS endpoint for an IN transfer. This API uses the 16-bit Endpoint registers to load the data.
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_ReadOutEP16()	Reads the specified number of bytes from the Endpoint buffer and places it in the system SRAM. Returns the number of bytes sent by the host. This API uses the 16-bit Endpoint registers to read the data.
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 Dp/Dm 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.
USBFS_Bcd_DetectPortType()	Determines if the host is capable of charging a downstream port.
USBFS_GetDeviceAddress()	Returns the currently assigned address for the USB device.
USBFS_EnableSofInt()	Enables interrupt generation when a Start-of-Frame (SOF) packet is received from the host.
USBFS_DisableSofInt	Disables interrupt generation when a Start-of-Frame (SOF) packet is received from the host.

void USBFS_Start(uint8 device, uint8 mode)

Description: This function performs all required initialization for the USBFS component. After this function call, the USB device initiates communication with the host by pull-up D+ line. This is the preferred method to begin component operation.

Note that global interrupts have to be enabled because interrupts are required for USBFS component operation.

PSoC 4200L devices: when USBFS component configured to DMA with Automatic Buffer Management, the DMA interrupt priority is changed to the highest (priority 0) inside this function.

PSoC 3/PSoC 5LP devices: when USBFS component configured to DMA with Automatic Buffer Management, the Arbiter interrupt priority is changed to the highest (priority 0) inside this function.

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_POWER_OPERATION	Enable or disable the voltage regulator depending on the power supply voltage configuration in the DWR tab. For PSoC 3/5LP devices, the VDDD supply voltage is considered and for PSoC 4A-L, the VBUS supply voltage is considered.

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_POWER_OPERATION	Enable or disable the voltage regulator depending on the power supply voltage configuration in the DWR tab. For PSoC 3/5LP devices, the VDDD supply voltage is considered and for PSoC 4A-L, the VBUS supply voltage is considered.

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 host sends double SET_CONFIGURATION requests with the same configuration number or changes alternate settings of the interface.

After configuration has been changed the OUT endpoints must be enabled and IN endpoint must be loaded with data to start communication with the host.

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.

It is useful to identify which alternate settings are active in 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 void USBFS_Stop_DMA(uint8 epNumber)

Description: This function stops DMA channel associated with endpoint. It is available when the Endpoint Buffer Management parameter is set to DMA.

Call this function when endpoint direction is changed from IN to OUT or vice versa to trigger DMA re-configuration when USBFS_LoadInEP() or USBFS_ReadOutEP() functions are called the first time.

Parameters: uint8 epNumber: The data endpoint number for which associated DMA channel is stopped. The range of valid values is between 1 and 8.

To stop all DMAs associated with endpoints call this function with USBFS_MAX_EP argument.

Return Value: None

Side Effects: None

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

Description: This function performs different functionality depending on the Component's configured Endpoint Buffer Management. This parameter is defined in the Descriptor Root section in Component Configure window.

Manual (Static/Dynamic Allocation): This function loads and enables the specified USB data endpoint for an IN data transfer.

DMA with Manual Buffer Management: Configures DMA for a data transfer from system SRAM to endpoint buffer. Generates request for a transfer.

DMA with Automatic Buffer Management:

1. Configure DMA. This is required only once, with parameter pData is not NULL.
2. Initiate DMA transaction on demand, with the pData pointer is NULL. Sets Data ready status: This generates the first DMA transfer and prepares data in endpoint buffer.

Parameters: uint8 epNumber: IN data endpoint number into which the data is loaded.

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

uint16 length: The number of bytes to transfer from the array and then send as a result of an IN request. The valid values are between 0 and 512 for Manual Buffer Management modes and up to 1023 for DMA with Automatic Buffer Management mode.

Return Value: None

Side Effects: None

void USBFS_LoadInEP16(uint8 epNumber, const uint8 pData[], uint16 length)

Description: This function performs different functionality depending on the Component's configured Endpoint Buffer Management. This parameter is defined in the Descriptor Root section in Component Configure window.

Manual (Static/Dynamic Allocation): This function loads and enables the specified USB data endpoint for an IN data transfer.

DMA with Manual Buffer Management: Configures DMA for a data transfer from system SRAM to endpoint buffer. Generates request for a transfer.

DMA with Automatic Buffer Management:

1. Configure DMA. This is required only once, with parameter pData is not NULL.
2. Initiate DMA transaction on demand, with the pData pointer is NULL. Sets Data ready status: This generates the first DMA transfer and prepares data in endpoint buffer.

Parameters: uint8 epNumber: IN data endpoint number into which the data is loaded.

const uint8 pData[]: Pointer to a data array from which the data into the endpoint space is copied. The allocated data array size must be greater for one byte then endpoint maximum packet size if its packet size is odd (data array size should be even). Also data array must be aligned to the boundary of two bytes for ensure correct 16-bit data access to its elements otherwise a hard fault condition will occur.

uint16 length: The number of bytes to transfer from the array and then send as a result of an IN request. The valid values are between 0 and 512 for Manual Buffer Management modes and up to 1023 for DMA with Automatic Buffer Management mode.

Return Value: None

Side Effects: None

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

Description: This function performs different functionality depending on the Component's configured Endpoint Buffer Management. This parameter is defined in the Descriptor Root section in Component Configure window.

Manual (Static/Dynamic Allocation): This function copies the specified number of bytes from endpoint buffer to system SRAM buffer. After data has been copied the endpoint is released to allow the host to write next data.

The function does not support partial data reads therefore all received bytes has to be read at once. The length argument must be equal to the number of actually received bytes from the host. Call function USBFS_GetEPCount() to get actual number of received bytes.

DMA with Manual Buffer Management: Configures DMA to transfer data from endpoint buffer to system SRAM buffer and generates a DMA request. The firmware must wait until the DMA completes the data transfer after calling the USB_ReadOutEP() API. For example, by checking endpoint state:

```
while (USB_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
{
}
```

The USBFS_EnableOutEP() has to be called to allow the host to write data into the endpoint buffer after DMA has completed transfer data from OUT endpoint buffer to system SRAM buffer.

The function does not support partial data reads therefore all received bytes has to be read at once. The length argument must be equal to the number of actually received bytes from the host. Call function USBFS_GetEPCount() to get actual number of received bytes.

DMA with Automatic Buffer Management: Configures DMA to transfer data from endpoint buffer to system SRAM buffer. Generally, this function should be called once to configure DMA for operation. Then use USBFS_EnableOutEP() to release endpoint to allow the host to write next data. The allocated buffer size and length parameter must be equal to endpoint maximum packet size.

Note We recommend calling it with length equal to endpoint maximum packet size to read all received data bytes. Use return value to get actual number of received bytes.

Parameters: uint8 epNumber: OUT data endpoint number from where the data is read.

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

uint16 length: The number of bytes to copy from the OUT endpoint buffer into system SRAM data array. The valid values are between 1 and 512 for Manual Buffer Management modes and up to 1023 for DMA with Automatic Buffer Management mode.

The additional restrictions for length parameter are provided in the function description.

Return Value: uint16: Number of bytes received

Side Effects: **DMA with Automatic Buffer Management**: The hardware can put more bytes in the SRAM buffer than the host writes into OUT endpoint. The function USBFS_GetEPCount() should be called to get actual number of bytes written by the host. For PSoC 3/PSoC 5LP the allocated SRAM buffer size for OUT endpoint should be (Max Packet Size + 2) if it is not multiple of 32 to void memory corruption.

uint16 USBFS_ReadOutEP16(uint8 epNumber, uint8 pData[], uint16 length)

Description: This function performs different functionality depending on the Component's configured Endpoint Buffer Management. This parameter is defined in the Descriptor Root section in Component Configure window.

Manual (Static/Dynamic Allocation): This function copies the specified number of bytes from endpoint buffer to system SRAM buffer. After data has been copied the endpoint is released to allow the host to write next data.

The function does not support partial data reads therefore all received bytes has to be read at once. The length argument must be equal to the number of actually received bytes from the host. Call function USBFS_GetEPCount() to get actual number of received bytes.

DMA with Manual Buffer Management: Configures DMA to transfer data from endpoint buffer to system SRAM buffer and generates a DMA request. The firmware must wait until the DMA completes the data transfer after calling the USB_ReadOutEP() API. For example, by checking endpoint state:

```
while (USB_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
{
}
```

The USBFS_EnableOutEP() has to be called to allow the host to write data into the endpoint buffer after DMA has completed transfer data from OUT endpoint buffer to system SRAM buffer.

The function does not support partial data reads therefore all received bytes has to be read at once. The length argument must be equal to the number of actually received bytes from the host. Call function USBFS_GetEPCount() to get actual number of received bytes.

DMA with Automatic Buffer Management: Configures DMA to transfer data from endpoint buffer to system SRAM buffer. Generally, this function should be called once to configure DMA for operation. Then use USBFS_EnableOutEP() to release endpoint to allow the host to write next data. The allocated buffer size and length parameter must be equal to endpoint maximum packet size.

Note We recommend calling it with length equal to endpoint maximum packet size to read all received data bytes. Use return value to get actual number of received bytes.

Parameters: uint8 epNumber: OUT data endpoint number from where the data is read.

uint8 pData[]: Pointer to a data array to which the data from the endpoint space is copied. The allocated data array size must be greater for one byte then endpoint maximum packet size if its packet size is odd (data array size should be even). Also data array must be aligned to the boundary of two bytes for ensure correct 16-bit data access to its elements otherwise a hard fault condition will occur.

uint16 length: The number of bytes to copy from the OUT endpoint buffer into system SRAM data array. The valid values are between 1 and 512 for Manual Buffer Management modes and up to 1023 for DMA with Automatic Buffer Management mode.

The additional restrictions for length parameter are provided in the function description

Return Value: uint16: Number of bytes received

Side Effects: **DMA with Automatic Buffer Management**: The hardware can put more bytes in the SRAM buffer than the host writes into OUT endpoint. The function USBFS_GetEPCount() should be called to get actual number of bytes written by the host.

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

uint8 USBFS_Bcd_DetectPortType (void)

Description: This function implements the USB Battery Charger Detection (BCD) algorithm to determine the type of USB host downstream port. This API is available only for PSoC 4 devices, and should be called when the VBUS voltage transition (OFF to ON) is detected on the bus. If the USB device functionality is enabled, this API first calls USBFS_Stop() API internally to disable the USB device functionality, and then proceeds to implement the BCD algorithm to detect the USB host port type. The USBFS_Start() API should be called after this API if the USB communication needs to be initiated with the host.

Note This API is generated only if the "Enable Battery Charging Detection" option is enabled in the "Advanced" tab of the component GUI.

Note API implements the steps 2-4 of the BCD algorithm which are – Data Contact Detect, Primary Detection, and Secondary Detection. The first step of BCD algorithm, namely, VBUS detection – shall be handled at the application firmware level.

Parameters: None

Return Value: uint8: The return value can be the following:

Return Value	Description
USBFS_BCD_PORT_SDP	Standard downstream port detected
USBFS_BCD_PORT_CDP	Charging downstream port detected
USBFS_BCD_PORT_DCP	Dedicated charging port detected
USBFS_BCD_PORT_UNKNOWN	Unable to detect charging port type (proprietary charger type)
USBFS_BCD_PORT_ERR	Error condition in detection process

Side Effects: None

uint8 USBFS_GetDeviceAddress(void)

Description: This function returns the currently assigned address for the USB device.

Parameters: None

Return Value: uint8: Returns the currently assigned address. Returns 0 if the device has not yet been assigned an address.

Side Effects: None

void USBFS_EnableSofInt(void)

Description: This function enables interrupt generation when a Start-of-Frame (SOF) packet is received from the host.

Parameters: None

Return Value: None

Side Effects: None

void USBFS_DisableSofInt(void)

Description: This function disables interrupt generation when a Start-of-Frame (SOF) packet is received from the host.

Parameters: None

Return Value: None

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

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 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.

< PSoC Creator Installation Folder >\psoc\content\cycomponentlibrary\CyComponentLibrary.cylib\USBFS_v3.0\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: Sends data to the host controller. A timeout is enabled. Reports the number of bytes successfully sent.

Parameters: const uint8 pData[]: Pointer to the bytes of data to be written.

uint16 size: Number of bytes to be written.

uint16 *count: Pointer to where the component writes the number of bytes actually written.

uint8 timeOut: Amount of time (in units of 10 milliseconds) that the component waits before indicating that communication have timed out.

Return Value: cystatus: Returns CYRET_SUCCESS or CYRET_TIMEOUT

Return Value	Description
CYRET_SUCCESS	Returned if one or more bytes were successfully written
CYRET_TIMEOUT	Returned if the host controller did not respond to the write in 10 * timeOut milliseconds

Side Effects: None

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

Description: Receives data from the host controller. A timeout is enabled. Reports the number of bytes successfully read.

Parameters: uint8 pData[]: Pointer to storage for the bytes of data to be read.

uint16 size: Number of bytes to read.

uint16 *count: Pointer to where the component writes the number of bytes actually read.

uint8 timeOut: Amount of time (in units of 10 milliseconds) that the component waits before indicating that communication have timed out.

Return Value: Returns CYRET_SUCCESS or CYRET_TIMEOUT

Return Value	Description
CYRET_SUCCESS	Returned if one or more bytes were successfully written
CYRET_TIMEOUT	Returned if the host controller did not respond to the write in 10 * timeOut milliseconds

Side Effects: None

USB Suspend, Resume, and Remote Wakeup

The USBFS component supports USB Suspend, Resume, and Remote Wakeup functionality. This functionality is tightly related with the user application, the USBFS component only provides the APIs to help user archive desired behavior. The additional processing required from the user application. The description of application processing is provided in the bottom of this section.

Normally a SOF packet (at full speed) is sent by the host every 1 ms, and this is what keeps the device awake. The host suspending the device by not sending anything to the device for 3 ms. To recognize this condition the bus activity has to be checked. It can be done use USBFS_CheckActivity() function. A suspended device may draw no more than 0.5 mA from Vbus therefore it is desired to put device into low power mode to consume less current. The USBFS_Suspend() function must be called before entering low power mode.

Note After entering low power mode, the data which is left in the IN or OUT endpoint buffers is not restored after wakeup and lost. Therefore, it should be stored in the SRAM for OUT endpoint or read by the host for IN endpoint before enter low power mode.

When the host wants to wake the device up after a suspend, it does so by reversing the polarity of the signal on the data lines for at least 20ms. The signal is completed with a low speed end of packet signal. The falling edge interrupt on Dp pin wakes up device and leads to exit low power mode when host drives resume. The USBFS_Resume() function must be called after exit low power mode.

To resume communication with the host the data endpoints has to be managed: the OUT endpoints have to be enabled and IN endpoints have to be loaded with data. For DMA with

Automatic Buffer Management all endpoints buffers has to be initialized again before making them available to the host.

Function	Description
USBFS_CheckActivity()	Returns the activity status of the bus since the last call of the function.
USBFS_Suspend()	Prepares the USBFS component to enter low power mode.
USBFS_Resume()	Prepares the USBFS component for active mode operation after exit low power mode.
USBFS_RWUEnabled()	Returns current remote wakeup status.

uint8 USBFS_CheckActivity(void)

Description: This function returns the activity status of the bus. It clears the hardware status to provide updated status on the next call of this function. It provides a way to determine whether any USB bus activity occurred. The application should use this function to determine if the USB suspend conditions are met.

Parameters: None

Return Value: uint8 cystatus: Status of the bus since the last call of the function.

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 prepares the USBFS component to enter low power mode.

The interrupt on falling edge on Dp pin is configured to wakeup device when the host drives resume condition. The pull-up is enabled on the Dp line while device is in low power mode. The supported low power modes are Deep Sleep (PSoC 4200L) and Sleep (PSoC 3/ PSoC 5LP).

Note For PSoC 4200L devices, this function should not be called before entering device Sleep mode. PSoC 4200L Sleep mode is only for CPU suspending.

Note After enter low power mode, the data which is left in the IN or OUT endpoint buffers is not restored after wakeup and lost. Therefore it should be stored in the SRAM for OUT endpoint or read by the host for IN endpoint before enter low power mode.

Parameters: None

Return Value: None

Side Effects: None

void USBFS_Resume(void)

Description: This function prepares the USBFS component for active mode operation after exit low power mode. It restores the component active mode configuration such as device address assigned previously by the host, endpoints buffer and disables interrupt on Dp pin.

The supported low power modes are Deep Sleep (PSoC 4200L) and Sleep (PSoC 3 / PSoC 5LP).

Note For PSoC 4200L devices, this function should not be called after exiting Sleep.

Note To resume communication with the host, the data endpoints must be managed: the OUT endpoints must be enabled and IN endpoints must be loaded with data. For DMA with Automatic Buffer Management, all endpoints buffers must be initialized again before making them available to the host.

Parameters: None

Return Value: None

Side Effects: None

uint8 USBFS_RWUEnabled(void)

Description: This function returns the current remote wakeup status.

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

Parameters: None

Return Value: Returns non-zero value if remote wakeup is enabled and zero otherwise.

Side Effects: None

Enter/Exit Low Power Mode Example

The following code is suggested to enter/exit low power mode after suspend condition has been detected (host does not send anything to device for 3 ms):

```
/* Prepare USBFS to enter low power mode. */
USBFS_Suspend();

/* Enter low power mode: DeepSleep for PSoC 4 or Sleep for PSoC 3/
* PSoC 5LP. The device wakes up when host drives resume on the bus.
* The wakeup source is PICU - USB Dp pin falling edge.
*/
#if (CY_PSOC4)
    CySysPmDeepSleep();
#else
    CyPmSaveClocks();
    /* Specify wakeup source explicitly. */
    CyPmSleep(PM_SLEEP_TIME_NONE, PM_SLEEP_SRC_PICU);
    CyPmRestoreClocks();
```

```
#endif /* (CY_PSOC4) */

/* Restore USBFS for active mode operation. */
USBFS_Resume();

/* Restore communication with host for IN and OUT endpoints. The code below
* intended to show common approach and depends on USBFS component
* configuration. It has to be modified to suite your needs.
*/

/* Register buffers for IN and OUT endpoints (only applicable for DMA with
* Automatic Buffer Management).
*/
#if (USBFS_EP_MANAGEMENT_DMA_AUTO)
    USBFS_LoadInEP (IN_EP,  bufferIn, IN_EP_LENGTH);
    USBFS_ReadOutEP(OUT_EP, bufferOut, OUT_EP_LENGTH);
#endif /* (USBFS_EP_MANAGEMENT_DMA_AUTO) */

/* Enable OUT endpoint. */
USBFS_EnableOutEP(OUT_EP);

/* Load IN enpoint. */
#if (USBFS_EP_MANAGEMENT_DMA_AUTO)
    USBFS_LoadInEP(IN_EP, NULL, IN_EP_LENGTH);
#else
     USBFS_LoadInEP(IN_EP, bufferIn, IN_EP_LENGTH);
#endif /* (USBFS_EP_MANAGEMENT_DMA_AUTO) */
```

Link Power Management (LPM) Support

The LPM feature is available only for PSoC 4200L devices. The LPM related APIs (provided below) are generated only if a **BOS descriptor** is present and **Enable LPM** option is enabled in the USB 2.0 Extension Descriptor under any of the device descriptors. For details about LPM please see *Link Power Management (LPM) - 7/2007* spec on the usb.org.

LPM Functions

Function	Description
USBFS_Lpm_GetBeslValue()	Returns the BESL value sent by the host.
USBFS_Lpm_RemoteWakeUpAllowed()	Return the remote wake up permission set for the device by host.
USBFS_Lpm_SetResponse()	Set the response for the received LPM token from host.
USBFS_Lpm_GetResponse()	Get the current response for the received LPM token from host.

uint32 USBFS_Lpm_GetBeslValue (void)

Description: This function returns the Best Effort Service Latency (BESL) value sent by the host as part of the LPM token transaction.

Parameters: None

Return Value: uint32: 4-bit BESL value received in the LPM token packet from the host

Side Effects: None

uint32 USBFS_Lpm_RemoteWakeUpAllowed (void)

Description: This function returns the remote wakeup permission set for the device by the host as part of the LPM token transaction.

Parameters: None

Return Value: uint32: 1-bit bRemoteWake field value received in the LPM token that enables device to wake the host upon application-specific event.

Return Value	Description
0	Remote wakeup is not allowed
1	Remote wakeup is allowed

Side Effects: None

void USBFS_Lpm_SetResponse(uint32 response)

Description: This function configures the response in the handshake packet the device has to send when an LPM token packet is received.

Parameters: uint32 response: type of response to return for an LPM token packet

Allowed response values:

Value	Description
USBFS_LPM_REQ_ACK	next LPM request will be responded with ACK
USBFS_LPM_REQ_NAK	next LPM request will be responded with NAK
USBFS_LPM_REQ_NYET	next LPM request will be responded with NYET

Return Value: None

Side Effects: None

uint32 USBFS_Lpm_GetResponse(void)

Description: This function returns the currently configured response value that the device will send as part of the handshake packet when an LPM token packet is received.

Parameters: None

Return Value: uint32 response: type of handshake response that will be returned by the device for an LPM token packet

Possible response values:

Return Value	Description
USBFS_LPM_REQ_ACK	next LPM request will be responded with ACK
USBFS_LPM_REQ_NAK	next LPM request will be responded with NAK
USBFS_LPM_REQ_NYET	next LPM request will be responded with NYET

Side Effects: None

Common 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.
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() function.
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() function.

Variable	Description
USBFS_deviceStatus	This is a two-bit variable that contains power status in the first (LSB) 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() function.

Report Storage Area

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>

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for the more details.

In order to add code to the macro callback present in the component’s generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will “uncomment” the function call from the component’s source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.
- Write the function implementation (in any user file).

Callback Function [4] / Associated Macro	Description
USBFS_cls.c	
uint8 USBFS_DispatchClassRqst_Callback(uint8 interface) / USBFS_DISPATCH_CLASS_RQST_CALLBACK	Handles class requests which are not handled by the component class handlers. This callback function provides the interface to which class request intended to and has to return USBFS_TRUE if request is handled or USBFS_FALSE otherwise.
USBFS_cdc.c	
uint8 USBFS_DispatchCDCClass_CDC_READ_REQUESTS_Callback(viod) / USBFS_DISPATCH_CDC_CLASS_CDC_READ_REQUESTS_CALLBACK	Handles CDC Class requests directed from device to host. This callback function has to return USBFS_TRUE if request is recognized and handled and USBFS_FALSE otherwise.
uint8 USBFS_DispatchCDCClass_CDC_WRITE_REQUESTS_Callback(void) / USBFS_DISPATCH_CDC_CLASS_CDC_WRITE_REQUESTS_CALLBACK	Handles CDC Class requests directed from host to device. This callback function has to return USBFS_TRUE if request is recognized and handled and USBFS_FALSE otherwise.
USBFS_msc.c	
uint8 USBFS_DispatchMSCClassRqst_Callback(void) / USBFS_DISPATCH_MSC_CLASS_RQST_CALLBACK	Handles MSC Class requests. This callback function has to return USBFS_TRUE if request is recognized and handled and USBFS_FALSE otherwise.
void USBFS_DispatchMSCClass_MSC_RESET_RQST_Callback(void) / USBFS_DISPATCH_MSC_CLASS_MSC_RESET_RQST_CALLBACK	Performs application-specific actions before response to MSC Reset request is sent to host.
USBFS_vnd.c	
uint8 USBFS_HandleVendorRqst_Callback(void) / USBFS_HANDLE_VENDOR_RQST_CALLBACK	Handles vendor requests which are not handled by the component handlers. This callback function has to return USBFS_TRUE if request is recognized and handled and USBFS_FALSE otherwise.
USBFS_midi.c	
void USBFS_MIDI_Init_Callback(void) / USBFS_MIDI_INIT_CALLBACK	Used in the USBFS_MIDI_InitInterface() function to perform additional application-specific actions.
void USBFS_MIDI_OUT_EP_Service_Callback(void) / USBFS_MIDI_OUT_EP_SERVICE_CALLBACK	Used in the USBFS_MIDI_OUT_Service() function to perform additional application-specific actions.
void USBFS_MIDI1_ProcessUsbOut_EntryCallback(void) / USBFS_MIDI1_PROCESS_USB_OUT_ENTRY_CALLBACK	Used in the USBFS_MIDI1_ProcessUsbOut() function to perform additional application-specific actions.
void USBFS_MIDI1_ProcessUsbOut_ExitCallback(void) / USBFS_MIDI1_PROCESS_USB_OUT_EXIT_CALLBACK	Used in the USBFS_MIDI1_ProcessUsbOut() function to perform additional application-specific actions.

4 The callback function name is formed by component function name optionally appended by short explanation and “Callback” suffix.

Callback Function [4] / Associated Macro	Description
void USBFS_MIDI2_ProcessUsbOut_EntryCallback(void) / USBFS_MIDI2_PROCESS_USB_OUT_ENTRY_CALLBACK	Used in the USBFS_MIDI2_ProcessUsbOut() function to perform additional application-specific actions.
void USBFS_MIDI2_ProcessUsbOut_ExitCallback(void) / USBFS_MIDI2_PROCESS_USB_OUT_EXIT_CALLBACK	Used in the USBFS_MIDI2_ProcessUsbOut() function to perform additional application-specific actions.
USBFS_hid.c	
void USBFS_FindReport_Callback(void) / USBFS_FIND_REPORT_CALLBACK	Used in the USBFS_FindReport() function to perform additional application-specific actions.
USBFS_audio.c	
void USBFS_DispatchAUDIOClass_AUDIO_READ_REQUESTS_Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_AUDIO_READ_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_AUDIO_WRITE_REQUESTS_Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_AUDIO_WRITE_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions
voidUSBFS_DispatchAUDIOClass_MUTE_CONTROL_GET_REQUEST_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_MUTE_CONTROL_GET_REQUEST_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_VOLUME_CONTROL_GET_REQUEST_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_VOLUME_CONTROL_GET_REQUEST_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions
void USBFS_DispatchAUDIOClass_OTHER_GET_CUR_REQUESTS_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_OTHER_GET_CUR_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_AUDIO_SAMPLING_FREQ_REQUESTS_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_AUDIO_SAMPLING_FREQ_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_MUTE_SET_REQUEST_Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_MUTE_SET_REQUEST_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_VOLUME_CONTROL_SET_REQUEST_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_VOLUME_CONTROL_SET_REQUEST_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_OTHER_SET_CUR_REQUESTS_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_OTHER_SET_CUR_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
void USBFS_DispatchAUDIOClass_AUDIO_CONTROL_SEL_REQUESTS_ Callback(void) / USBFS_DISPATCH_AUDIO_CLASS_AUDIO_CONTROL_SEL_REQUESTS_CALLBACK	Used in the USBFS_DispatchAUDIOClassRqst() function to perform additional application-specific actions.
USBFS_dvr.c	

Callback Function [4] / Associated Macro	Description
void USBFS_EP_0_ISR_EntryCallback(void) / USBFS_EP_0_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_0_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_0_ISR_ExitCallback(void) / USBFS_EP_0_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_0_ISR() interrupt handler to perform additional application-specific actions.
USBFS_pm.c	
void USBFS_DP_ISR_EntryCallback(void) / USBFS_DP_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_DP_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_DP_ISR_ExitCallback(void) / USBFS_DP_ISR_EXIT_CALLBACK	Used at the end of the USBFS_DP_ISR() interrupt handler to perform additional application-specific actions.
USBFS_episr.c	
void USBFS_EP_1_ISR_EntryCallback(void) / USBFS_EP_1_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_1_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_1_ISR_ExitCallback(void) / USBFS_EP_1_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_1_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_2_ISR_EntryCallback(void) / USBFS_EP_2_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_2_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_2_ISR_ExitCallback(void) / USBFS_EP_2_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_2_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_3_ISR_EntryCallback(void) / USBFS_EP_3_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_3_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_3_ISR_ExitCallback(void) / USBFS_EP_3_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_3_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_4_ISR_EntryCallback(void) / USBFS_EP_4_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_4_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_4_ISR_ExitCallback(void) / USBFS_EP_4_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_4_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_5_ISR_EntryCallback(void) / USBFS_EP_5_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_5_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_5_ISR_ExitCallback(void) / USBFS_EP_5_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_5_ISR() interrupt handler to perform additional application-specific actions.

Callback Function [4] / Associated Macro	Description
void USBFS_EP_6_ISR_EntryCallback(void) / USBFS_EP_6_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_6_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_6_ISR_ExitCallback(void) / USBFS_EP_6_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_6_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_7_ISR_EntryCallback(void) / USBFS_EP_7_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_7_ISR() interrupt handler to perform additional application-specific actions
void USBFS_EP_7_ISR_ExitCallback(void) / USBFS_EP_7_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_7_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_8_ISR_EntryCallback(void) / USBFS_EP_8_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_8_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_8_ISR_ExitCallback(void) / USBFS_EP_8_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_8_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_SOF_ISR_EntryCallback(void) / USBFS_SOF_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_SOF_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_SOF_ISR_ExitCallback(void) / USBFS_SOF_ISR_EXIT_CALLBACK	Used at the end of the USBFS_SOF_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_BUS_RESET_ISR_EntryCallback(void) / USBFS_BUS_RESET_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_BUS_RESET_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_BUS_RESET_ISR_ExitCallback(void) / USBFS_BUS_RESET_ISR_EXIT_CALLBACK	Used at the end of the USBFS_BUS_RESET_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_ARB_ISR_EntryCallback(void) / USBFS_ARB_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_ARB_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_ARB_ISR_ExitCallback(void) / USBFS_ARB_ISR_EXIT_CALLBACK	Used at the end of the USBFS_ARB_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_ARB_ISR_Callback(uint8 epNumber, uint8 epStatus) / USBFS_ARB_ISR_CALLBACK	Called in the loop which services active arbiter interrupt sources from endpoints in the USBFS_ARB_ISR() interrupt handler. This callback function provides the endpoint which has pending arbiter interrupt and its arbiter status register (ARB_EPx_SR) which contains only enabled arbiter interrupt sources.

Callback Function [4] / Associated Macro	Description
void USBFS_EP_DMA_DONE_ISR_EntryCallback(void) / USBFS_EP_DMA_DONE_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_EP_DMA_DONE_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_DMA_DONE_ISR_ExitCallback(void) / USBFS_EP_DMA_DONE_ISR_EXIT_CALLBACK	Used at the end of the USBFS_EP_DMA_DONE_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_EP_DMA_DONE_ISR_Callback(void)/ USBFS_EP_DMA_DONE_ISR_CALLBACK	Used in the_EP_DMA_DONE_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_LPM_ISR_EntryCallback(void) / USBFS_LPM_ISR_ENTRY_CALLBACK	Used at the beginning of the USBFS_LPM_ISR() interrupt handler to perform additional application-specific actions.
void USBFS_LPM_ISR_ExitCallback(void) / USBFS_LPM_ISR_EXIT_CALLBACK	Used at the end of the USBFS_LPM_ISR() interrupt handler to perform additional application-specific actions.

Sample Firmware Source Code

PSoC Creator provides many code examples that include schematics and example code in the Find Code Example 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 Code Example” 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 deviations:

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.
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.
19.7	Advisory	A function should be used in preference to a function-like macro.	Deviations with function-like macros to allow for more efficient code. The component incorporates DMA component. Macros with arguments are used to achieve high speed of DMA set up .

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

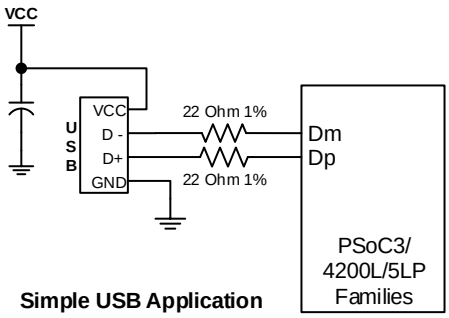
The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

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 4200L		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Default USBFS	8365	222	5589	215	5591	241
Maximum, with HID,CDC, Audio, MIDI and Bootloader	19278	469	11108	496	11218	495

Functional Description

The following diagram shows a simple bus-powered USB application with Dp and Dm pins from the PSoC device.



Simple USB Application

USBFS Basic Workflow In Different Modes

USBFS supports three primary modes for Endpoint Buffer Management:

- Manual (Static Allocation / Dynamic Allocation)
- DMA with Manual Buffer Management (Static Allocation)
- DMA with Automatic Buffer Management

The mode is chosen in the **Device Descriptor** tab by selecting the **Descriptor Root** node in the tree view.

The component provides all the necessary functions to manage data flow from host to the device by using OUT endpoints and from device to the host by using IN endpoints. To access the IN data endpoints, the function USBFS_LoadInEP() is provided. For data OUT endpoints, the function USBFS_ReadOutEP() is provided. Note that 16-bit accesses are also supported. Refer to 16-bit Endpoint Access API section for more details.

Manual

Manual endpoint management mode means that firmware intervention is needed to move data between endpoint buffers and SRAM buffers. Hence the CPU is used to move the data between them. The SRAM buffer is separate from the endpoint buffer and can be used by the application.

For IN direction, the USBFS_LoadInEP() function returns when the data is written into the endpoint buffer and the endpoint is exposed to the host to be read.

For OUT direction, the USBFS_ReadOutEP() function returns when the data is read from the endpoint buffer and the endpoint is exposed to the host to be written.

Data write into the IN endpoint:

```
/* Check that IN endpoint buffer is empty before write data. */
if (USBFS_IN_BUFFER_EMPTY == USBFS_GetEPState(IN_EP))
{
    /* Write data into IN endpoint buffer.
    * Data is written after function returns and endpoint is available to be
    * read by host.
    */
    USBFS_LoadInEP(IN_EP, buffer, length);
}
```

Data read from OUT endpoint:

```
/* Check if OUT endpoint buffer contain data to be read. */
if (USBFS_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
{
    /* Read data from OUT endpoint buffer.
     * Data is read after function returns and endpoint is available to be
     * written by host.
     */
     USBFS_ReadOutEP(OUT_EP, buffer, length);
}
```

DMA with Manual Buffer Management

DMA with Manual Buffer Management is almost identical to Manual mode with the difference that DMA moves data between endpoint buffer and SRAM buffer instead of the CPU. The USBFS_LoadInEP() and USBFS_ReadOutEP() functions only initiate the DMA data transfer but do not wait for its completion:

For IN direction, the transfer is handled without user intervention because the arbiter interrupt triggers when the IN endpoint buffer is full (signaling that the DMA has completed “writing” the data) and the endpoint is exposed to the host to be read.

For OUT direction, it is the user’s responsibility to check that the DMA has completed “reading” the data from the endpoint buffer before accessing this data. After the DMA transfer is complete, the arbiter interrupt triggers and change the endpoint state from “buffer full” to “buffer empty”.

Data write into the IN endpoint:

```
/* Check that IN endpoint buffer is empty before write data. */
if (USBFS_IN_BUFFER_EMPTY == USBFS_GetEPState(IN_EP))
```

```
{
    /* Write data into IN endpoint buffer.
    * The DMA request is generated to write data into endpoint buffer.
    * When DMA is done the arbiter interrupt fires and makes endpoint available to
    * be read by host.
    */
    USBFS_LoadInEP(IN_EP, buffer, length);
}
```

Data read from OUT endpoint:

```
/* Check if OUT endpoint buffer contain data to be read. */
if (USBFS_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
{
    /* Read data from OUT endpoint buffer.
     * DMA Manual: DMA request is generated to read data from endpoint buffer.
     * When DMA is done arbiter interrupt fires and updates endpoint status that
     * buffer is empty. Also it makes endpoint available to be written by host.
     */
     USBFS_ReadOutEP(OUT_EP, buffer, length);

    /* DMA Manual: wait until DMA is done reading data from OUT endpoint buffer.
    */
    while (USBFS_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
    {
    }

    /* Enable OUT endpoint and allow host to write new data. */
    USBFS_EnableOutEP(OUT_EP);
}
```

DMA with Automatic Buffer Management

DMA auto management (DMA with Automatic Buffer Management) is quite different from the modes described above. This mode allows the DMA hardware to manage all data movement between the endpoint buffers and the SRAM buffers. The implementation details are as follows.

1. Each endpoint buffer size is 32 bytes and the rest of the USB block buffer (size of 512 bytes) is a common area which acts as a FIFO for the endpoint that to which the host currently communicates.
2. When the endpoint buffer is full, the data comes into the FIFO (common area). This allocation allows servicing endpoint sizes greater than the USB block buffer size, because the USB block buffer is used as the infrastructure to manage data transfers and the actual data is stored in the system SRAM memory.
3. The DMA reading from or writing into the endpoint buffer happens in 32 byte chunks per request. Transfers greater than 32 bytes require multiple DMA requests to complete. The requests are generated automatically by the USB block when the host starts a communication with an endpoint.
4. For OUT transfers, the common area stores data which has not yet been read by the DMA. For IN transfers the DMA tries to fill the common area as much as possible to

provide the USB block with data to send. The DMA also pre-loads 32 bytes into the endpoint buffer before allowing the host to start reading it. This action adds time for the DMA to write next chunks of the data while the host reads the pre-loaded data. In case the DMA fails to load data in time (before the host reads it), the old data which resides in the common area will be sent on the bus.

In PSoC 3/PSoC 5LP devices, the DMA engine is capable of managing data transfers in chunks of 32 bytes using multiple requests from the USB block. Therefore functions USBFS_LoadInEP() and USBFS_ReadOutEP configure the DMA and the transfer is handled automatically by the hardware.

PSoC 4200L devices have a different DMA engine to the one present in PSoC 3/PSoC 5LP devices. The usage is the same as PSoC 3/PSoC 5LP, but the architecture necessitates a background firmware interaction to achieve the same functionality.

1. The DMA channels for USBFS component are set up with 2 chained descriptors which transfers 32 bytes each.
2. After the descriptors have executed, they are invalidated and the following DMA request causes a DMA error interrupt to occur (there is no valid descriptor to execute when a request comes). To service this interrupt source, the DMA interrupt handler must find the channel that caused the interrupt and call its ISR handler provided by the USBFS component (the DMA engine has a single interrupt for all channels).
3. Inside this handler, the channel descriptors source is updated for IN transfer and destination for OUT transfer. DMA execution with the updated descriptors is then triggered using firmware.

Note The DMA interrupt service time is critical for correct USBFS component operation, especially for IN traffic. Therefore the component sets the DMA interrupt priority to the highest level (refer to section Interrupt Service Routine). The application must manage other interrupt priorities, DMA channel priorities and preemption to provide enough bandwidth for USBFS DMA channels operation.

The DMA auto management also requires registering SRAM buffers after the device has been enumerated. The provided SRAM buffer is used by the component and is accessed whenever communication to an endpoint occurs. For IN endpoint, ensure that the transfer is completed before modifying the buffer content. For OUT endpoint, received data must be copied from the buffer or handled before enabling the OUT endpoint to avoid data re-writing.

Initialization of IN and OUT endpoints buffers:

```
/* Register SRAM buffers for IN and OUT EP endpoints. */
USBFS_LoadInEP (IN_EP,  bufferIn, length);
USBFS_ReadOutEP(OUT_EP, bufferOut, length);
```

Data write into the IN endpoint:

```
/* Make sure that IN endpoint buffer is empty before write data. */
while (USBFS_IN_BUFFER_EMPTY != USBFS_GetEPState(IN_EP))
{
```

```
}

/* The DMA request is generated to write 32 bytes into IN endpoint buffer. When
 * DMA is done the arbiter interrupt fires and makes endpoint available to be
 * read by host. As soon as host starts reading next DMA transfer starts to fill
 * common area.
 * The DMA keeps transferring data until all data has been transferred. The
 * DMA can be paused if common area is full.
 */
USBFS_LoadInEP(IN_EP, NULL, length);
```

Data read from OUT endpoint:

```
/* Check if OUT endpoint buffer contains data to be read */
if (USBFS_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))
{
    /* The data is available in the buffer has to be copied from it or handled
     * before enable OUT endpoint. This step is required to avoid data corruption.
     * Because after endpoint is enabled the buffer will be updated as soon as new
     * transfer to OUT endpoint starts.
     */
    length = USBFS_GetEPCount(OUT_EP);

    /* Copy data into the local buffer to handle it later. */
    memcpy(appBufferOut, bufferOut, length);

    /* Enable OUT endpoint and allow host to write new data. */
    USBFS_EnableOutEP(OUT_EP);
}
```

Note Using loops (for example, `while (USBFS_OUT_BUFFER_FULL == USBFS_GetEPState(OUT_EP))` for waiting of data is dangerous in case of changing the endpoint direction during change of alternate settings. Changing of endpoint direction during waiting in such loop could cause to infinite waiting, because exit event never happens. Empty IN buffer is interpret as full OUT buffer and vice versa.

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 you are creating a self-powered device, you must connect a GPIO pin to VBUS through a resistive network and write firmware to monitor the status of the GPIO (see VBUS Monitoring). You can use the USBFS_Start() and USBFS_Stop() API routines to control the Dp and Dm 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 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

For other configuration class requests, refer to the specific configuration under Additional USBFS Configurations.

16-bit Endpoint Access API

The USBFS_LoadInEP16() and USBFS_ReadOutEP16() functions provide 16-bit access to the endpoint data registers. These functions are available when Generate 16-bit endpoint access APIs option is enabled in the custom dialog's **Advanced** tab (only applicable for PSoC 4200L devices).

The 16-bit access allows reading or writing two bytes instead of one byte (as is the case when 8-bit access is used by USBFS_LoadInEP() / USBFS_ReadOutEP() functions). This makes 16-bits access function work two times faster. However, the data buffers for 16-bits functions must be allocated using the following rules:

- Allocated data buffer size must be a multiple of 2 regardless of the transfer length. In cases when the endpoint maximum packet size is odd, the extra element allocated should be ignored by the application. For example, when the maximum endpoint packet size is 63 bytes, the allocated data buffer must be 64 bytes.
- Allocated data buffer must be aligned to 2-bytes boundary, otherwise hard fault can occur during 16-bit access. Typically each compiler provides attributes to define the required alignment.

The following is an example of loading 63 bytes into an IN endpoint using the 16-bit function with a GCC compiler:

```
/* IN endpoint number */
#define IN_EP           (1u)
/* Number of bytes to transfer for IN request */
#define TRANSFER_LENGTH (63u)
/* Buffer size for 16-bit function */
```

```
#define BUFFER_SIZE     (64u)

/* Allocate and align the RAM buffer */
uint8 buffer[BUFFER_SIZE] __attribute__ ((aligned (2)));

/* Load data into the IN endpoint buffer to be transferred to the host */
USBFS_LoadInEP16(IN_EP, buffer, TRANSFER_LENGHT);
```

Interrupt Service Routine

The USBSF component supports interrupts from a number of sources. The number of interrupts vary depending on the component configuration and selected device.

PSoC 3/PSoC 5LP devices consume up to 13 interrupts, some of which are mandatory for correct component operation and some which are optional. Each interrupt source has separate interrupt handlers in PSoC 3/PSoC 5LP.

PSoC 4200L devices consume up to 3 interrupts – labeled Low, Medium and High. The configured interrupt sources and their interrupt handlers are mapped to these interrupts. Refer to the Interrupts Tab section for the list of available interrupt sources. Note that Low, Medium and High interrupts are just the names assigned for the USB interrupts and has no relation to the priority assigned to them.

The list of USBFS component interrupt handlers is provided below:

- USBFS_EP_0_ISR – triggered to handle control transfers directed to endpoint 0. This interrupt handler is mandatory for component operation.
- USBFS_EP_X_ISR – triggered at the end of a transfer directed to endpoints 1–8 (where X is the endpoint number from 1 to 8). For IN endpoint, it means that the data loaded into endpoint buffer is read by the host. For OUT endpoint it means that data is available in the endpoint buffer. These are mandatory for component operation.

  For PSoC 4200L, when Endpoint Buffer Management mode is set to **DMA with Automatic Buffer Management**, the DMA can still transfer data from OUT endpoint buffer to SRAM buffer when USBFS_EP_X_ISR occurs. The arbiter interrupt is used to identify the end of the transfer.
- USBFS_BUS_RESET_ISR – triggered when bus reset condition is detected. This interrupt handler is mandatory for component operation.
- USBFS_LPM_ISR – triggered after LPM (Link Power Management) request is confirmed by ACK response (applicable for PSoC 4200L). The BOS descriptor should present in device descriptor tree and **Enable LPM** option has to be enabled in BOS descriptor to invoke this interrupt handler. The user can use it to read the BESL value send by the host and then decide whether to enter low power mode or not.
- USBFS_ARB_ISR – triggered when arbitration is needed between SIE (Serial Interface Engine) and CPU or DMA access to an endpoint buffer. It is mandatory for component

operation in DMA with Manual or Automatic Buffer Management modes. It is not needed in Manual mode. The following interrupt sources trigger an arbiter interrupt for each endpoint:

- IN endpoint buffer is full – triggers when IN endpoint buffer is loaded with data. It is used by the component to release the IN endpoint to be read by the host.
- Endpoint DMA grant – triggers to indicate completion of the DMA transfer corresponding to the raised DMA request. It is used by the component to update the status of OUT endpoint when the data has been copied from endpoint buffer to SRAM. It is only used in the Manual DMA endpoint memory management mode.
- Endpoint buffer overflow – triggers to indicate that overflow condition happened for a data endpoint. It is only applicable in DMA with Automatic Buffer Management. This interrupt source is not enabled by the component and should never occur.
- Endpoint buffer underflow – triggers to indicate that underflow condition happened for a data endpoint. It is only applicable in DMA with Automatic Buffer Management. This interrupt source is not enabled by the component and should never occur.
- Endpoint error in transaction – The error in transaction bit is set whenever an error is detected. For an IN transaction, this indicates a no response from HOST scenario. For an OUT transaction, this represents a PID error / CRC error / bit-stuff error scenario. This interrupt source is not enabled by the component and should never occur.
- Endpoint DMA terminated – triggers to indicate that DMA transfer has to be terminated. This interrupt source only applicable in DMA with Automatic Buffer Management for PSoC 4200L devices. It is used by the component to terminate a DMA transfer and update the status of OUT endpoint when the data has been copied from endpoint buffer to SRAM.

- USBFS_EP_DMA_DONE_ISR – triggered when DMA completes a data transfer for any of the endpoints. This interrupt handler is applicable only in DMA with Automatic Buffer Management mode for PSoC 3/PSoC 5LP devices. It is mandatory for the component operation and is used to override erroneous hardware behavior for IN endpoints. It is intended for internal component use only.
- USBFS_EPX_DMA_DONE_ISR – triggered when DMA completes transferring a chunk of data into the endpoint buffer (where X is endpoint number from 1 to 8). This interrupt handler is applicable only in DMA with Automatic Buffer Management mode for PSoC 4200L devices. It is mandatory for the component operation and is intended for internal component usage only.
- USBFS_SOF_ISR – triggered when SOF packet is received. This interrupt handler is not used for component operation and is disabled by default. Refer to section Enable SOF interrupt for information how to enable it.

- USBFS_DP_ISR – triggered when host drives resume on the DP line from high to low. This is only one interrupt, which could wake up device from the suspend mode. See USB Suspend, Resume, and Remote Wakeup for details.

Each interrupt handler has callbacks inside it, which allows adding custom code. The macro callbacks are listed in the Macro Callbacks section for more details.

Interrupt priority change

The USBFS component modifies the priority of interrupts that are critical for its operation under certain configurations.

PSoC 4200L Devices

When the USBFS component is configured in DMA with Automatic Buffer Management mode, the DMA interrupt service of the USBFS channels is critical for correct component operation. Therefore USBFS component changes DMA interrupt priority to the highest (priority 0) inside USBFS_Start() function. Note that is interrupt is not available in the Design Wide Resources file on the Interrupts tab.

PSoC 3/PSoC 5LP Devices

When the USBFS component is configured in DMA with Automatic Buffer Management mode, the arbiter interrupt (USBFS_arb_int) indicates completion of DMA request service. It is critical for the system to set the priority of this interrupt higher than the priority of the USBFS_ep[0..8] interrupts. Therefore USBFS component changes arbiter interrupt priority to the highest (priority 0) inside USBFS_Start() function.

Clock Selection

The USB hardware block requires specific configuration of the system clocks. The clocks can be configured through the PSoC Creator Design-Wide Resources Clock Editor. Refer to the Quick Start section for clock requirements on different devices.

Link Power Management (LPM)

Link Power Management (LPM) is a USB low power mode feature that provides more flexibility in terms of features than the existing resume mode. The feature is available only in PSoC 4200L device. See the LPM related APIs in the Link Power Management (LPM) Support section.

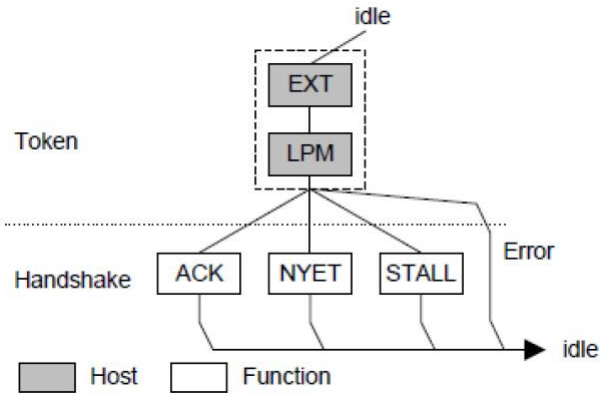
LPM is a new power management feature for USB2.0 which is similar to the existing suspend/resume, but has transitional latencies of tens of microseconds between power states, instead of three to greater than 20 millisecond latencies of the USB2.0 suspend/resume.

USB2.0 Power states are re-arranged as below with the introduction of LPM. The existing power states are re-named with LPM. They are

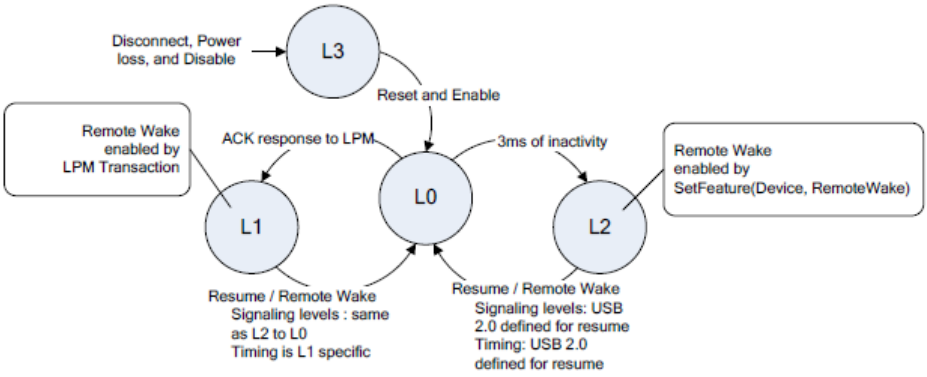
- L0 (On)

- L1 (Sleep) -- Newly Introduced State in LPM
- L2 (Suspend)
- L3 (Powered-Off)

When a host is ready to transition a device from L0 to a deeper power savings state, it issues an LPM transaction to the device. The device function responds with an ACK if it is ready to make the transition or a NYET (Not Yet) if it is not currently ready to make the transition (usually because it is has data pending for the host). A Device will transmit a STALL handshake if it does not support the requested link state. If the device detects errors in either of the token packets or does not understand the protocol extension transaction, no handshake is returned.



Transitions between different power modes are shown as follows:



PSoC 4200L has a separate interrupt for LPM request (see Interrupt Service Routine section).

LPM interrupt has a callback function **USBFS_LPM_ISR_EntryCallback().** This is called after LPM request is ACKed only. You can use this callback function to set a global variable to identify received LPM request in the application.

LPM related descriptor (USB 2.0 Extension Descriptor) has attributes named Baseline BESL and deep BESL (see USB2.0 Extension Descriptor section). Using these attributes inform the host about time it needs to wake up from 2 different suspend modes. Baseline BESL is meant for fast

wake up time and less energy efficiency, whereas Deep BESL translates to slower wake up time and more energy efficiency.

An application should use baseline BESL and deep BESL values to decide which low power mode enters to. For example when received BESL is less than baseline BESL leave device in the Active mode, when it is between baseline BESL and deep BESL put device into the Deep Sleep mode and when it is greater than deep BESL put device into Hibernate mode.

Note Device will restart after hibernate mode and the USBFS component needs re-initialization at the application level. USBFS_Suspend() and USBFS_Resume() APIs do not support Hibernate mode. During initialization, the component does not drive Dp signals and reset condition will be detected if the host drive resume condition less then device wake up time (see USBFS_LPM_PSoC4 code example). The application should ensure that the device will resume within the time defined in the BESL value of LPM request.

The following code shows the basic flow for entering/exiting low power mode after receiving an LPM request (see USBFS_LPM_PSoC4 code example):

```
/* Global flag for LPM request detection. */
uint8 beslValue;

/* LPM request ISR callback. It updates BESL value in global
* beslValue variable and set flag that LPM request received.
*/
void USBFS_LPM_ISR_EntryCallback(void)
{
    /* Get BESL value and try to enter low power mode. */
    beslValue  = USBFS_Lpm_GetBeslValue();
    activeMode = FALSE;
}

/* Main routine. */

/* Check if device is started after hibernate. */
if (CySysPmGetResetReason() != CY_PM_RESET_REASON_WAKEUP_HIB)
{
    /* Normal start. */
}
else
{
    /* Start after hibernate: need to restore component register and internal
    * structures.
    */
}

/* Active mode operation after start. */
activeMode = TRUE;

/* Main application loop */
for (;;)
{
    if (FALSE != activeMode)
    {
         /* Active mode run. */
```

```
        }
        else
        {
            /* Handle enter/exit from low power mode. */
        }
    }

    /* Section to proceed with low power node*/

    /* Check what mode we could enter based on received BESL value in LPM request from
    * host. */
    if ((beslValue >= BESL_BASELINE) && (beslValue < BESL_DEEP_MODE))
    {
        /* Prepare USBFS to enter low power mode. */
        USBFS_Suspend();

        /* Put device into DeepSleep mode. */
        CySysPmDeepSleep();

        /* Restore USBFS for active mode operation. */
        USBFS_Resume();

        /* Restore communication with host for OUT endpoint. */

        /* Enable OUT endpoint. */
        USBFS_EnableOutEP(OUT_EP);

        /* Active mode operation. */
        activeMode = TRUE;
    }
    else if (beslValue >= BESL_DEEP_MODE) /* We have time to resume from hibernate. */
    {
        /* Prepare USBFS to enter low power mode. */
        USBFS_Suspend();

        /* Put device into hibernate mode. */
        CySysPmHibernate();

        /* The exit for Hibernate is reset. */
    }
    else /* Resume time is too small keep device in active mode. */
    {
        /* Continue active mode operation. */
        activeMode = TRUE;
    }
```

Note that USBFS_Start() API configures the device to ACK for LPM requests. You may change the device response to LPM request using USBFS_Lpm_SetResponse() API.

LPM request also allows performing remote wake up. You can check the value of remote wake up allowance by using USBFS_Lpm_RemoteWakeUpAllowed() API. Based on the returned value of this API, the application should decide whether to perform host remote wake up procedure.

Additional USBFS Configurations

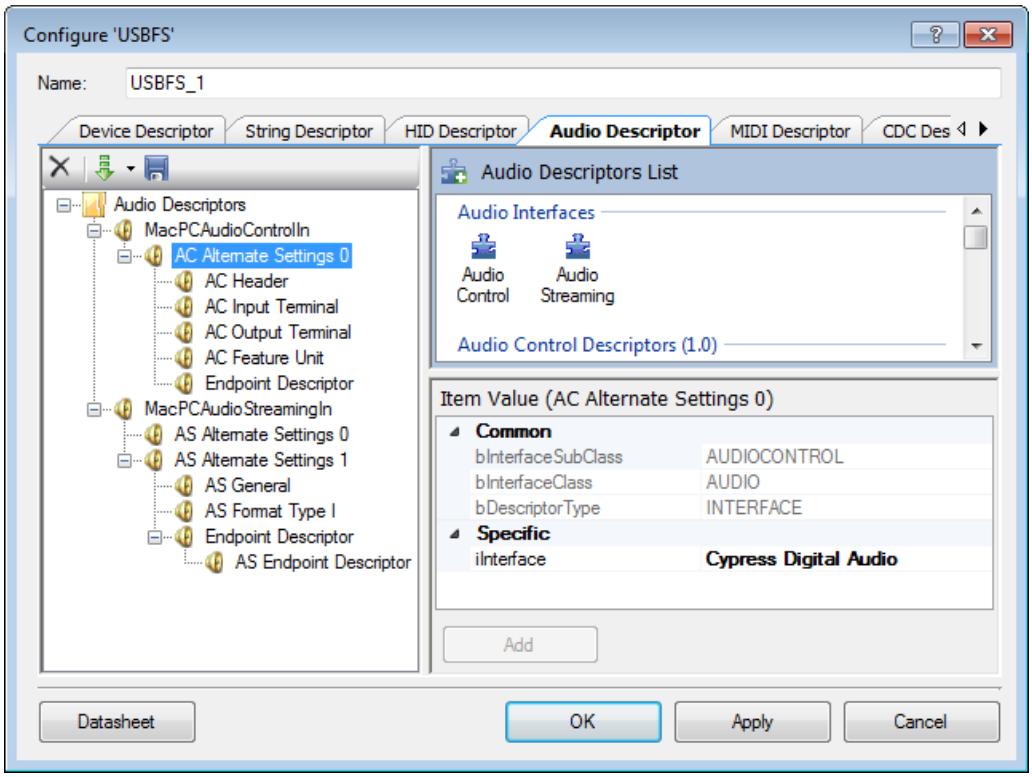
In addition to the base configuration, the USBFS component can be configured into the following options:

- USBFS Audio
- USBFS MIDI
- USBUART (CDC)
- USBFS MSC

USBFS Audio

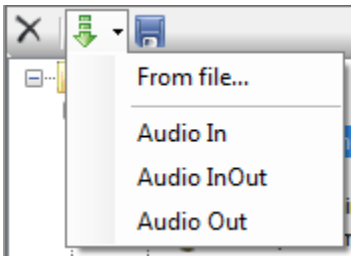
The USBFS component provides support for Audio class descriptors. The USBFS Audio interface is implemented according to the *Universal Serial Bus Device Class Definition for Audio Devices 1.0 and 2.0* specifications.

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



Import Audio Descriptor

The **Import** button allows you to quickly import the Audio descriptors from saved templates.



In the drop-down list you can choose one of the templates or choose an existing descriptor template from a file location. The predefined descriptors are available in the folder:

The template options immediately load the selected HID report. The **From file** option will open an audio descriptor that was created by the USBFS Component.

Audio Descriptors List

This area allows you to add Audio descriptors. Detailed information on the Audio descriptors is available in the Universal Serial Bus Device Class Definition for Audio.

Item Value

This area allows you select a value that is appropriate for the currently selected Audio Descriptor item. The parameters in these windows are context based and will vary depending upon the item value selected in the Audio Descriptors List window.

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. 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 it again or by using of Rename context menu item.

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.

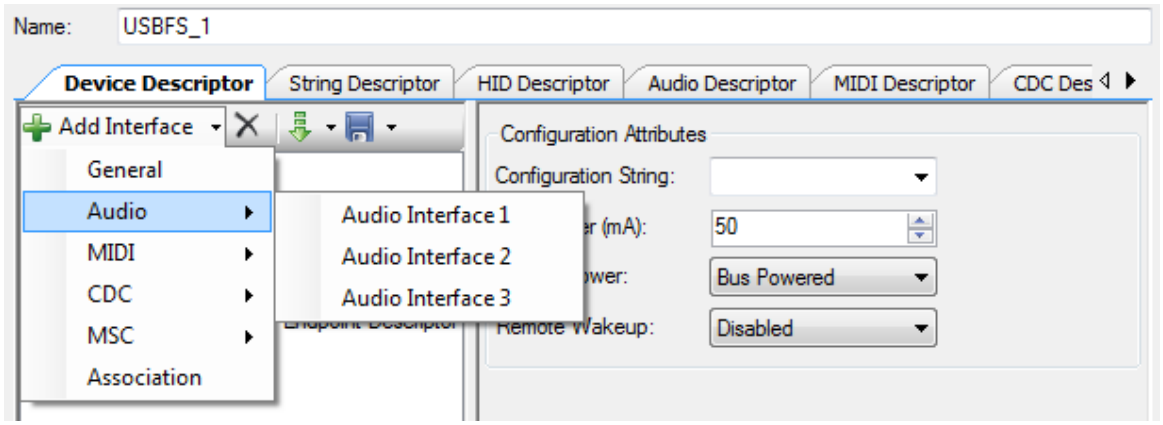
To Add Standard AS Isochronous Synch Endpoint Descriptor

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.

USBFS Audio 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.

Audio Class Request

This section describes the requests supported by the USBFS component according to Universal Serial Bus Device Class Definition for Audio Devices 1.0. 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

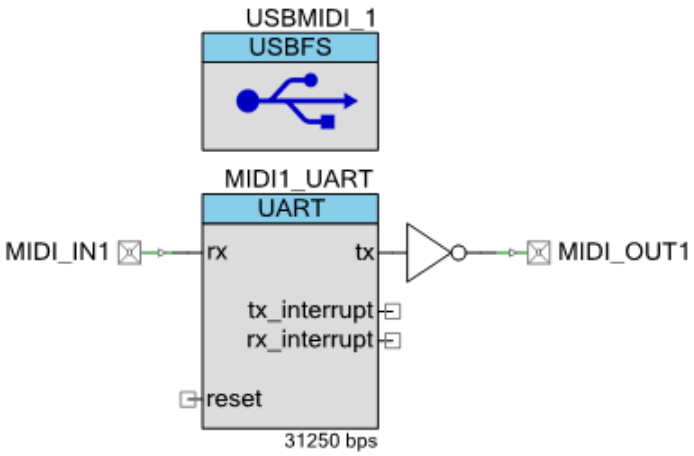
USBFS MIDI provides support for communicating with external MIDI equipment. It also provides support for the USB device class definition for MIDI devices. You can use this component to add MIDI I/O capability to a standalone device, or to implement MIDI capability for a host computer or mobile device 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.

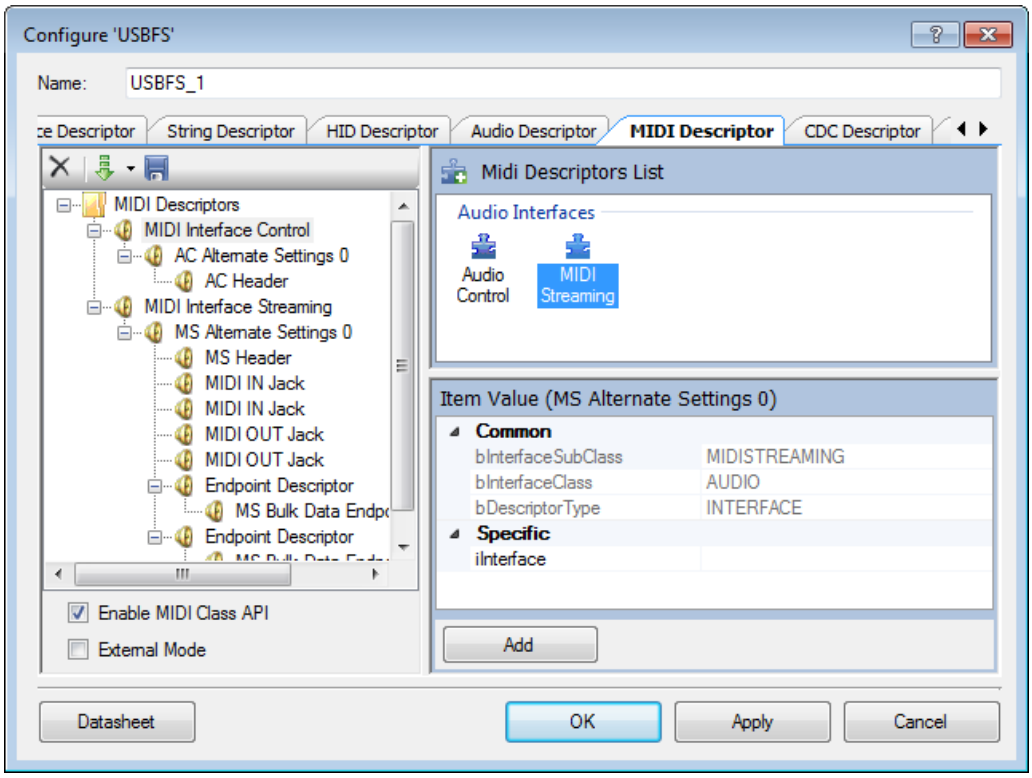
The PSoC Creator Component Catalog contains a Schematic Macro implementation of a MIDI interface. The macro consists of instances of the UART component with the hardware MIDI interface configuration (31.25 kbps, 8 data bits) and a USBFS component with the descriptors configured to support MIDI devices.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

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



Enable MIDI Class API

This parameter enables generation of MIDI Class APIs. The list of generated APIs is provided in the section USBFS MIDI .

External Mode

This option determines the source of the MIDI messages as external (option is enabled) and internal (option is disabled). External means that the UART component is used to send and receive MIDI messages to external MIDI equipment. The USBFS component supports up to two UART instances. Their names have to be "MIDI1_UART" and "MIDI2_UART" because USBFS component call their APIs. When option is disabled the local switches and sensors can be used to create MIDI messages for the host. For more information refer to the USBFS MIDI Functional Description section.

MIDI Descriptors List

This window allows you to add MIDI descriptors. Detailed information on the MIDI descriptors is available in the Universal Serial Bus Device Class Definition for MIDI Devices Revision 1.0.

Item Value

This window allows you select a value that is appropriate for the currently selected MIDI Descriptor item. The parameters in these windows are context based and will vary depending upon the item value selected in the MIDI Descriptors List window.

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

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

   ```
   <PSoC Creator Installation Folder>\psoc\content\cycomponentlibrary\
   CyComponentLibrary.cylib\USBFS_v3.0\Custom\template\
   ```

3. Go to the **Device Descriptor** tab. Remove MIDI Interface Control and MIDI Interface Streaming interface descriptors from Configuration Descriptor.
4. Add interfaces: MIDI Interface Control and MIDI Interface Streaming to Configuration Descriptor. Close USBMIDI_1 component customizer by pressing OK button.
5. Clone existing UART instance with NOT gate, MIDI_IN1 and MIDI_OUT1 pins using Copy/Paste method.
6. Change name of cloned elements, to MIDI2_UART, MIDI_IN2, and MIDI_OUT2 accordingly.
7. Set size of MIDI1_UART and MIDI2_UART Rx/Tx buffers to the recommended using appropriate customizer (see Interrupt Priority section).

To Add MIDI Descriptors

1. Select the **MIDI Descriptors** root item in the tree on the left.
2. Under **MIDI Descriptors List** on the right, select either the **Audio Control** or **MIDI Streaming** interface.
3. 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 clicking on it again or by using of Rename context menu item.

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.

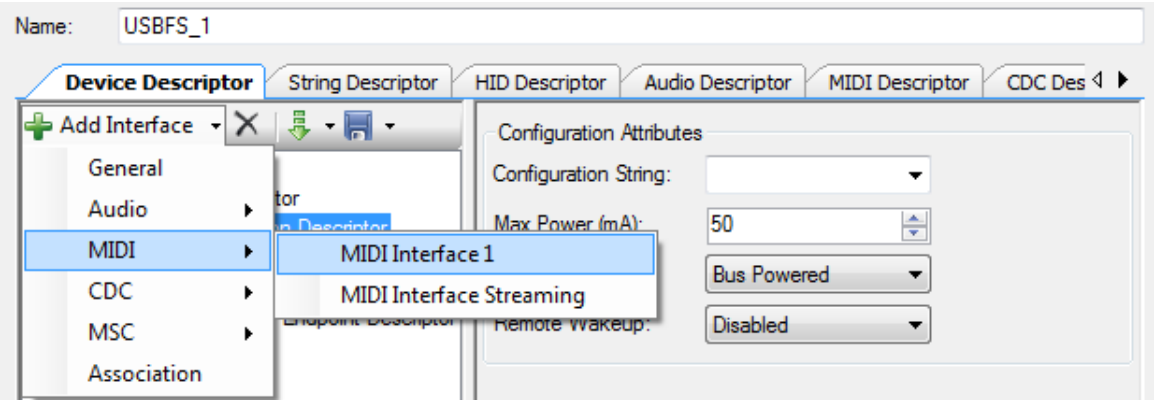
To Add Standard MS Bulk Data Endpoint Descriptor

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 Functions

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_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_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*.

void USBMIDI_MIDI_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: The priority of the UART RX ISR should be higher than UART TX ISR. So the API changes the priority of the UARTs' TX and RX interrupts.

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

In PSoC 3, if this function is called from an ISR, you must declare this function as re-entrant so that different variable storage space is created by the compiler. This is automatically taken care for PSoC 4200L and PSoC 5LP devices by the compiler.

Parameters: None

Return Value: None

Side Effects: None

void USBMIDI_MIDI_OUT_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 Global Variables

<table>
<tr><th>Variable</th><th>Description</th></tr>
<tr><td>USBMIDI_midiInBuffer</td><td>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.</td></tr>
<tr><td>USBMIDI_midiOutBuffer</td><td>Output endpoint buffer with a length equal to MIDI OUT EP Max Packet Size. This buffer is used by the USBMIDI_MIDI_OUT_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.</td></tr>
<tr><td>USBMIDI_midiInPointer</td><td>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.</td></tr>
<tr><td>USBMIDI_midi_in_ep</td><td>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.</td></tr>
<tr><td>USBMIDI_midi_out_ep</td><td>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.</td></tr>
<tr><td>USBMIDI_MIDI1_InqFlags
USBMIDI_MIDI2_InqFlags</td><td>These optional variables are allocated when External Mode is enabled. The following flags help to detect and generate responses for SysEx messages.
<table>
<tr><th>Flag</th><th>Description</th></tr>
<tr><td>USBMIDI_INQ_SYSEX_FLAG</td><td>Non-real-time SysEx message received.</td></tr>
<tr><td>USBMIDI_INQ_IDENTITY_REQ_FLAG</td><td>Identity Request received. You should clear this bit when an Identity Reply message is generated.</td></tr>
</table>
</td></tr>
</table>

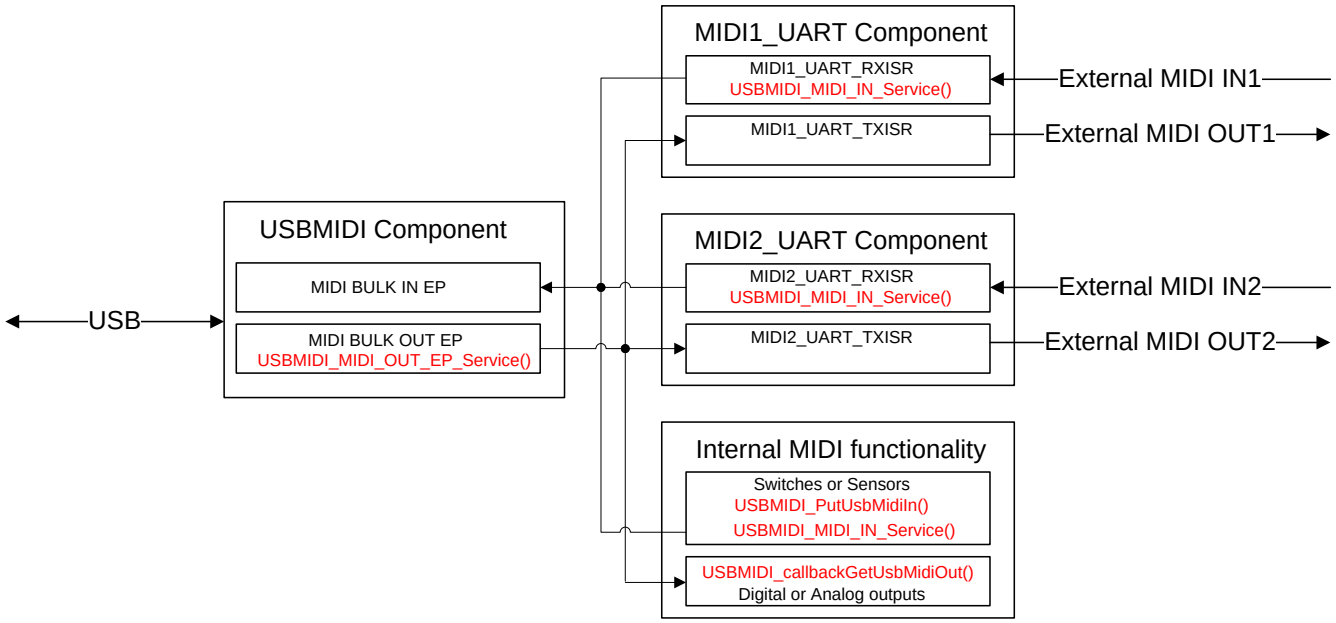
USBFS MIDI Functional Description

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

Figure 2. 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_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

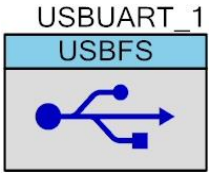
The data received from the host is serviced inside the MIDI BULK OUT EP ISR. When you select a small UART **TX Buffer Size**, the code waits for the UART transmit operation to complete and continues filling the TX buffer. The priority of the Interrupt Service Routine for the TX portion of the UART should 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 255.

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.



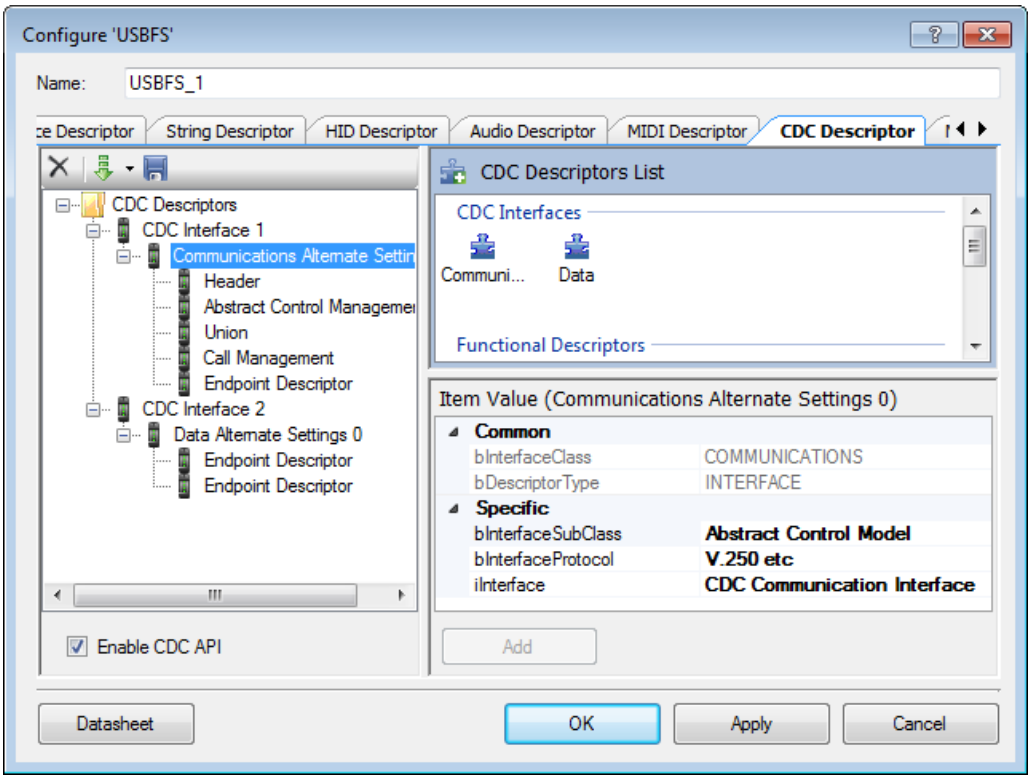
To start a USBUART-based project, drag the USBUART Schematic Macro labeled ‘USBUART (CDC Interface)’ from the Component Catalog onto your design. This macro has already been configured to function as a CDC device. See the Component Parameters section of this datasheet for information about modifying 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\<PSOC_NAME>\<INSTANCE_NAME>_cdc.inf

See the USBFS_UART code example for detailed steps on how to install a driver.

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



Enable CDC API

This option enables generation of CDC APIs. The list of generated APIs is provided in the section USBUART (CDC) .

CDC Descriptors List

This **area** allows you to add CDC descriptors. Detailed information on the CDC descriptors is available in the Universal Serial Bus Class Definitions for Communication Devices.

Item Value

This window allows you select a value that is appropriate for the currently selected CDC Descriptor item. The parameters in these windows are context based and will vary depending upon the item value selected in the CDC Descriptors List window.

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. Click **Add** to add the descriptor to the tree on the left.
4. You can rename the **CDC Interface x** title by selecting a node and clicking on it again or by using of Rename context menu item.

To Add Functional Descriptors

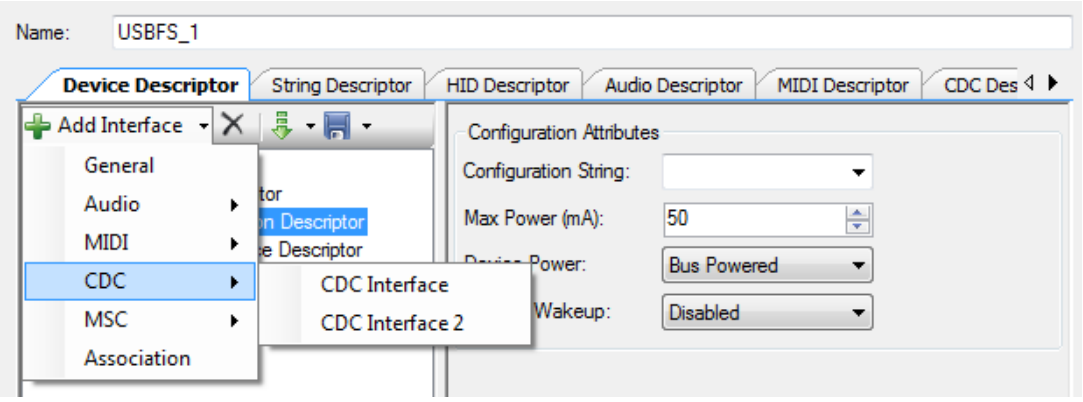
1. Select the appropriate **Communications Alternate Settings x** item in the tree on the left.
2. Under the **CDC Descriptors List** on the right, select one of the items under **Functional Descriptors** as appropriate.
3. Under **Item Value**, enter the appropriate values under **Specific**.
4. Click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

The following high-level APIs are available when the **Enable CDC 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
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
USBUART_SendSerialState()	Sends the serial state notification to the host using the interrupt endpoint

Function	Description
USBUART_GetSerialState()	Returns the current serial state value
USBUART_SetComPort()	If multiple COM ports are instantiated, this function selects which COM port user wish to address.
USBUART_GetComPort()	If multiple COM ports are instantiated, this function returns the current selected COM port that the user is addressing.
USBUART_NotificationIsReady()	Returns a nonzero value if the component is ready to send more notification data to the host

void USBUART_CDC_Init(void)

Description: This function initializes the CDC interface to be ready to receive data from the PC. The API set active communication port to 0 in the case of multiple communication port support. 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(0 == USBUART_GetConfiguration())
{
}
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. The function does not support the partial data reads therefore all received bytes has to be read at once. The length argument must be equal to the number of actually received bytes from the host. Call function USBUART_GetCount() to get actual number of received bytes.

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 length must be equal 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 1-byte data packet from the buffer. This function must not be called if more than 1 byte is received, because call of this function for more than 1 byte received case could lead to unpredicted results. Use USBUART_GetCount() API to get number of received bytes.

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

void USBUART_SendSerialState (uint16 serialState)

Description: Sends the serial state notification to the host using the interrupt endpoint for the COM port selected using the API SetCOMPort().The USBUART_NotificationIsReady() API must be called to check if the Component is ready to send more serial state to the host. The API will not send the notification data if the interrupt endpoint Max Packet Size is less than the required 10 bytes.

Parameters: uint16 serialState: 16-bit value that will be sent from the device to the host as SERIAL_STATE notification using the IN interrupt endpoint. Refer to revision 1.2 of the CDC PSTN Subclass specification for bit field definitions of the 16-bit serial state value.

Return Value: None

Side Effects: None

uint16 USBUART_GetSerialState (void)

Description: This function returns the current serial state value for the COM port selected using the API SetCOMPort().

Parameters: None

Return Value: uint16:16-bit serial state value. Refer to revision 1.2 of the CDC PSTN Subclass specification for bit field definitions of the 16-bit serial state value.

Side Effects: None

int8 USBUART_NotificationIsReady (void)

Description: This function returns a nonzero value if the Component is ready to send more notifications to the host; otherwise, it returns zero. The function should be called before sending new notifications when using USBUART_SendSerialState() to ensure that any previous notification data has been already sent to the host.

Parameters: None

Return Value: uint8: If the buffer can accept new data (endpoint buffer not full), this function returns a nonzero value. Otherwise, it returns zero.

Side Effects: None

void USBUART_SetComPort(uint8 comPortNumber)

Description: This function allows the user to select from one of the two COM ports they wish to address in the instance of having multiple COM ports instantiated though the use of a composite device. Once set, all future function calls related to the USBUART will be affected. This addressed COM port can be changed during run time.

Parameters: uint8 comNumber: Contains the COM interface the user wishes to address. Value can either be 0 or 1 since a maximum of only 2 COM ports can be supported. Note that this COM port number is not the COM port number assigned on the PC side for the UART communication. If a value greater than 1 is passed, the function returns without performing any action.

Return Value: None

Side Effects: None

int8 USBUART_GetComPort(void)

Description: This function returns the current selected COM port that the user is currently addressing in the instance of having multiple COM ports instantiated though the use of a composite device.

Parameters: None

Return Value: uint8: Returns the currently selected COM port. Value can either be 0 or 1 since a maximum of only 2 COM ports can be supported. . Note that this COM port number is not the COM port number assigned on the PC side for the UART communication

Side Effects: None

USBUART (CDC) 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. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
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. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
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. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
USBUART_serialStateBitmap[]	Contains the 16-bit serial state value that was sent using the USBUART_SendSerialState() API. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
USBUART_cdcDataInEp[]	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. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
USBUART_cdcDataOutEp[]	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. It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.
USBUART_cdcCommInInterruptEp[]	Contains the IN interrupt endpoint number used for sending serial state notification to the host. It is initialized after a SET_CONFIGURATION request based on a user descriptor. It is used in the CDC API USBUART_SendSerialState().It is an array of 2 elements for COM port 1 and COM port 2 for MultiCOM port support. In case of 1 COM port, data is in 0 element.

USBUART (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

Supported by the USBUART component PSTIN subclass specific notification:

Class Notification	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

Note Use USBUART_SendSerialState() and USBUART_GetSerialState() API to work with SERIAL_STATE notification.

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 USBUART (CDC) Functions list earlier in the datasheet).

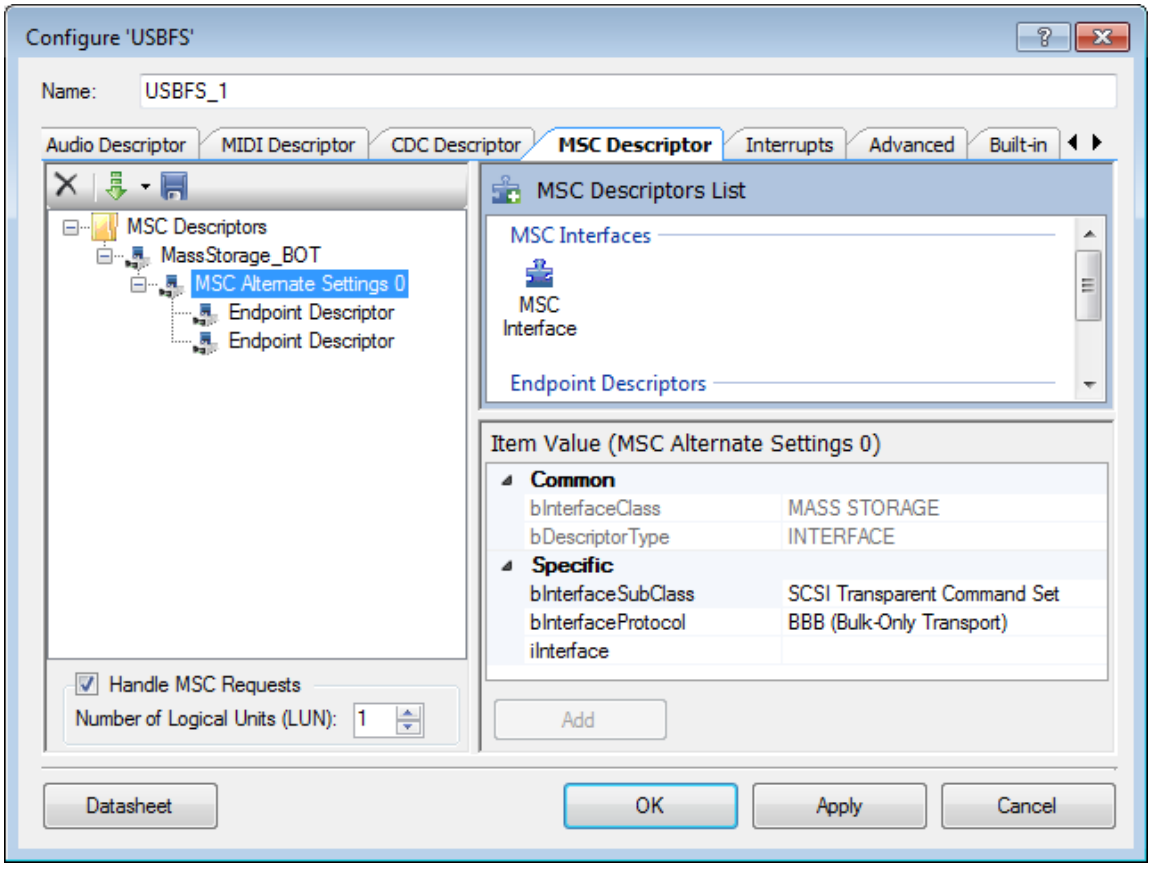
Table 1. API Migration

CE60246 API	USBFS v2.0+ API	Changes Required in Migration
void USBUART_Init (void)	void USBUART_CDC_Init (void)	API name change.
uint8 USBUART_bGetRxCount (void)	uint16 USBUART_GetCount (void)	API name change. Return value changed from uint8 to uint16.
void USBUART_ReadAll (uint8* pData)	uint16 USBUART_GetAll (uint8* pData)	API name change. Return value changed from void to uint16.
void USBUART_Write (uint8 *pData, uint8 bLength)	void USBUART_PutData (const uint8* pData, uint16 length)	API name change. Length parameter type changed from uint8 to uint16.
uint8 USBUART_bTxIsReady (void)	uint8 USBUART_CDCIsReady (void)	API name change.

Note The table assumes the component name is “USBUART.”

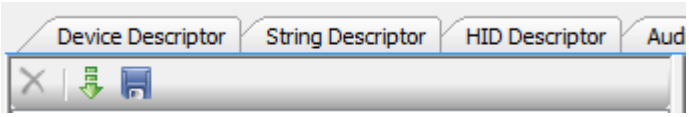
USBFS MSC

The USBFS component provides limited support for mass storage class (MSC) descriptors. To add and configure mass storage interface descriptors, open the Configure USBFS dialog and click the **MSC Descriptor** tab.



Import MSC Descriptor

The **Import** button allows you to quickly import the MSC descriptors from saved template. The single MSC descriptors template is provided in the MassStorage_BOT.msc file.



It is possible create MSC manually instead of using template. But it is recomented to start from the template.

MSC Descriptors List

This area allows you to add MSC descriptors. Detailed information on the MSC descriptors is available in the Universal Serial Bus Device Class Definition for Mass Storage.

Item Value

This area allows you select a value that is appropriate for the currently selected MSC Descriptor item. The parameters in these windows are context based and will vary depending upon the item value selected in the MSC Descriptors List window.

To Add MSC Descriptors

1. Select the **MSC Descriptors** root item in the tree on the left.
2. Under the **MCS Descriptors List** on the right, select the **MSC Interface Descriptor**.
3. Click **Add** to add the descriptor to the tree on the left.

   You can rename the **MSC Interface x** title by selecting a node and clicking on it again or by using of Rename context menu item.

To Add MSC Endpoint Descriptors

1. Select the appropriate **MSC Alternate Settings x** item in the tree on the left.
2. Under the **MSC 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.

Handling MSC Request

The component provides option **Handle MSC Requests** in the **MSC Descriptor** tab which defines

This option determines if component handles the MSC requests or it becomes user responsibility. When this option is enabled the component handle MSC requests defined in the Universal Serial Bus Mass Storage Class Bulk-Only Transport specification and provides functions listed in the MSC section.

Otherwise component allows users to implement callback function which handles MSC requests. The callback name is USBFS_DispatchMSCClassRqst_Callback(). Refer to the Macro Callbacks section for details. For example of class request handler implementation find USBFS_DispatchMSCClassRqst() function in the USBFS_msc.c file or any other dispatch class request function.

Number of Logical Units (LUN)

This parameter specifies the number of logical units that is supported by the Mass Storage device. This filed is available only if Handling MSC Request option is enabled.

MSC Functions

The following functions and callback are available when the Handling MSC Request option is enabled.

Function	Description
USBFS_MSC_SetLunCount()	Sets the number of logical units.
USBFS_MSC_GetLunCount()	Returns the number of logical units.

void USBFS_MSC_SetLunCount(uint8 lunCount)

Description: This function sets the number of logical units supported in the application. The default number of logical units is set in the component customizer.

Parameters: lunCount: Count of the logical units. Valid range is between 1 and 16.

Return Value: None

Side Effects: None

uint8 USBFS_MSC_GetLunCount(void)

Description: This function returns the number of logical units.

Parameters: None

Return Value: uint8: Number of the logical units.

Side Effects: None

MSC Class Request

The MSC Reset request typically requires application service. The component allows user to implement callback function USBFS_DispatchMSCClass_MSC_RESET_RQST_Callback() to add application part of this request service. The response to request is not sent to the host until application processing completes. Refer to Macro Callbacks section for details how to enable macro callback.

Component Debug Window

PSoC Creator allows you to view debug information about components in your design. Each component window lists the memory and registers for the instance. For detailed hardware registers descriptions, refer to the appropriate device technical reference manual.

To open the Component Debug window:

1. Make sure the debugger is running or in break mode.
2. Choose **Windows > Components…** from the **Debug** menu.

3. In the Component Window Selector dialog, select the component instances to view and click **OK**.

The selected Component Debug window(s) will open within the debugger framework. Refer to the "Component Debug Window" topic in the PSoC Creator Help for more information.

Resources

USB is implemented as a fixed-function block.

Configuration	Resource Type			
	USBFS Fixed Blocks	Clocks		Pins
		PSoC 3 / PSoC 5LP	PSoC 4200L	
All Configurations	1	IMO, Clock Doubler, 100 kHz ILO	IMO, ILO	3

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 PSoC 3/5LP	4.35	–	5.25	V
		USB configured, USB regulator enabled PSoC4200L	4.5	–	5.5	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
$I_{USB\_Suspended}$	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

Parameter	Description	Conditions	Min	Typ	Max	Units
		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		4	–	20	ns
Tf	Transition fall time		4	–	20	ns
TR	Rise/fall time matching		90%	–	111%	
V_{CRS}	Output signal crossover voltage		1.3	–	2	V

Component Errata

This section lists known problems with the component.

Cypress ID	Version	Problem	Workaround
227772	All	This issue is applicable to the PSoC 3 and PSoC 5LP USB component operating in “DMA with Manual Buffer” mode. The component may intermittently fail to load the IN endpoints completely if the USB device is overloaded with too many requests. When this problem occurs, the IN endpoint is no longer exposed to the host and reports that the IN buffer is full.	It is recommenced to use other modes of operation (“Manual” or “DMA with Automatic Buffer Management”) if you face this problem.

Cypress ID	Version	Problem	Workaround

231992	All	This issue is applicable to PSoC 3 and PSoC 5LP devices. The USBFS_Stop() function does not turn off all hardware. As a result, the device increases power consumption after the USBFS_Start() and USBF_Stop() sequence.	After calling USBFS_Stop(), disable input receivers and clear DP interrupt:
			`/* Ensure single-ended disable bits are high (PRT15.INP_DIS[7:6])`
			`* (input receiver disabled). */`
			`USBFS_DM_INP_DIS_REG |= (uint8) USBFS_DM_MASK;`
			`USBFS_DP_INP_DIS_REG |= (uint8) USBFS_DP_MASK;`
			`/*Clean up Dp interrupt to prevent current leakage and unexpected wake up*/`
			`(void) USBFS_Dp_ClearInterrupt();`
			`CyIntClearPending( USBFS_DP_INTC_VECT_NUM);`

252866	3.0	This issue applicable to PSoC 4200L USB component operating in “DMA with Automatic Buffer Management” mode.
		The component could corrupt memory after several USB plug in / plug out, due to false triggering of DMA during USB plug out.
			Add check that DMA is not triggered after last DMA burst.
			Check should be added in USBFS_episr.c file in each USBFS_EP<num>_DMA_DONE_ISR()endpoint function used in project. <num> mean number of end point (1-8).
			Possible implementation:
			`void USBFS_EP<num>_DMA_DONE_ISR(void)`
			`{`
			`static uint8 dmaDone = 0u;`
			`/* Manage data elements which remain to transfer. */`
			`if (0u == USBFS_DmaEpBurstCnt[USBFS_EP<num>])`
			`{`
			`/* Adjust length of last burst. */`
			`}`
			`else`
			`{`
			`dmaDone = 0u;`
			`}`
			`if (0u == dmaDone)`
			`{`
			`/* Advance source for input endpoint or destination for output endpoint. */`
			`/* Enable DMA to execute transfer as it was disabled because there were no valid descriptor. */`
			`}`
			`if (0u == USBFS_DmaEpBurstCnt[USBFS_EP<num>])`
			`{`
			`dmaDone = 1;`
			`}`
			`else`
			`{`
			`--USBFS_DmaEpBurstCnt[USBFS_EP<num>];`
			`}`
			`}`

Cypress ID	Version	Problem	Workaround
274441	3.0, 3.10	The USBFS Component fails to recover endpoint operation after the host issues CLEAR_FEATURE request to the endpoint.	There is no workaround. Contact Cypress to obtain an updated Component.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
3.0.b	Datasheet edits.	Added errata item 274441.
3.0.a	Datasheet edits.	Various formatting and edits to make the document more clear. Removed note about data for PSoC 4200L devices being preliminary. Added errata item 252866 to document issue with repeated USB plug in and plug out attempts. Added errata item 231992 to document power increase issue after USB_Start USB_Stop sequence.
3.0	Version 3.0 of the USBFS component is not completely backward compatible with the previous versions due to a change in the register access format to follow a common firmware architecture for PSoC 3/PSoC 4200L/ PSoC 5LP. The unused variables were removed or renamed to match the naming conversion.	The implementation in previous component versions to access registers was inconvenient, and not scalable to support the newer PSoC 4 family devices with USB. The backward compatibility exception is only for the user section code in the USB component ISRs. The compilation error will appear if old variables were used in the user section code. The user code would have to be updated to fix the compilation error.
	Change control flow for handling OUT endpoint in the DMA with Manual Buffer Management mode. The OUT endpoint is not enabled in the arbiter interrupt anymore after USBFS_ReadOutEP() is called.	The condition that DMA has completed transfer data from OUT endpoint buffer to SRAM buffer can fail when short USB transfer comes in. The USBFS_EnableOutEP() has to be called to allow host to write data into the endpoint buffer after DMA has completed transfer data from OUT endpoint buffer to SRAM buffer.
	Added support for PSoC 4200L devices. The new features specific for PSoC 4200L are BCD and LPM.	

Version	Description of Changes	Reason for Changes / Impact
	Added support for multiple COM ports.	
	Added support for CDC SERIAL_STATE notification.	
	Added support for Mass Storage Class (MSC) descriptor configuration and MSC request handling.	
	Changed initial drive state of D+ pin to high for PSoC3 / PSoC5LP devices.	Remove glitch on D+ line while device wakeups from Sleep when host drives resume.
	Macro Callbacks section update.	Removed SOF callback: USBFS_SOF_ISR_InterruptCallback() Added new macro callbacks for SOF: USBFS_SOF_ISR_EntryCallback() and USBFS_SOF_ISR_ExitCallback()
	Updated MIDI functions names. ▪ New names: USBFS_MIDI_Init() and USBFS_MIDI_OUT_Service() ▪ Old names: USBFS_MIDI_EP_Init() and USBFS_MIDI_OUT_EP_Service()	Made MIDI function more consistent with other component functions.
	Datasheet updates	Improve description in sections: ▪ USB, Suspend, Resume, and Remote Wakeup ▪ Endpoint Buffer Management ▪ Interrupt Service Routine Added USBFS Basic Workflow in Different Modes section. Added Component Errata section to document issue with the PSoC 3 and PSoC 5LP USB component operating in “DMA with Manual Buffer” mode.
2.80.a	Datasheet update.	Added Macro Callbacks section.
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.

Version	Description of Changes	Reason for Changes / Impact
	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.
	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.	

Version	Description of Changes	Reason for Changes / Impact
	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.	
	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.

Version	Description of Changes	Reason for Changes / Impact
	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.
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 advertises 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 advertises 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 advertises 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.