AN56377

PSoC® 3 and PSoC 5 USB Transfer Types

Author: Robert Murphy
Associated Project: Yes
Associated Part Family: CY8C3xxx, CY8C5xxx
Software Version: PSoC® Creator™ 2.0
Associated Application Notes: See Related Resources

Abstract

AN56377 describes the four USB transfer types: Interrupt, Bulk, Isochronous, and Control. It then shows how to configure PSoC® 3 and PSoC 5 to perform each of these transfers. Code examples are also included for specific considerations, including vendor commands for custom USB functionality, and to use DMA for faster data throughput. This application note assumes advanced level knowledge of USB. For an introduction to USB, see AN57294.

Contents

Introduction 2
Application Note Requirements 2
USB Transfers 2
  USB Transfer Structure 3
  Transfer Composition 3
USB Transfer Types 5
  Interrupt Transfers 5
Bulk Transfers 7
  Isochronous Transfers 8
  Synchronization of Isochronous Transfers: 10
  Feedback and Feedforward 11
Control Transfers 12
USB Requests 13
Bus Bandwidth Management 16
USB Error Correction and Detection 16
  CRC Check 17
  PID Check 17
  Data Toggle 17
Error Detection in PSoC 3 and PSoC 5 19
PSoC Logical Transfer Modes (PSoC 3 Only) 20
Project Files 21
Project 1: Implementing Multiple Transfer Types 21
  Configuring the Project 21
Testing the Project 27
Project 2: Implementing Vendor Commands 28
  Configuring the Project 28
  Testing the Project 31
Project 3: Increasing USB Throughput with DMA on PSoC 3 32
  Configuring the Project 32
  Testing the Project 34
Summary 35
Related Resources 35
  Application Notes 35
  Additional Information 35
About the Author 35
Appendix A: INF for AN56377 36
Appendix B: main.c for Project 1 37
Appendix C: USBFS_1_vnd.c for Project 2 39
Appendix D: main.c for Project 2 40
Appendix E: main.c for Project 3 41
Document History 42
Worldwide Sales and Design Support 43

Introduction

Most people think of USB communication simply as the transfer of data between a device and a PC. This may be accurate from a high level perspective; however, when you design and debug a USB application, you must think about USB communication in more detail.

USB communication occurs through a series of transfers. The USB specification defines four different transfer types: Control, Interrupt, Bulk, and Isochronous. Depending on what the end application/device is, one transfer type may be a better option than others.

Along with transfer types, the USB specification also defines certain commands that a USB device needs to understand and respond to. However, you can define your own USB commands that can be used to simply turn on/off a LED indicator or change the entire configuration of an application. These custom commands are called vendor commands and are useful in providing direct control of the device, independent of the general 'bit hose' used when streaming data.

Additionally, when you select the proper transfer type for your application, you must ensure that the data arrived at its destination is intact and that the information is valid. There are multiple error checks and error corrections implemented in a USB system. There are also handshakes, which provide a closed loop feedback between the device and host with regard to the result of the transfer, the result being whether the transfer was complete or incomplete. Knowing about the different handshakes, error corrections, and error checks is important to help understand how USB works and to aid in debugging a USB design.

The purpose of this application note is to build upon your USB knowledge by teaching you how to choose and implement the proper transfer type in an application. USB bandwidth is limited and each transfer has advantages and disadvantages. Choosing the proper transfer type can increase the performance of your application and conserve bus bandwidth. This application note also shows you how to leverage the powerful DMA in PSoC 3 to move data around the PSoC to obtain greater throughput than using the CPU alone. Finally, this application note shows you how to implement vendor commands in your application to provide direct control of various peripherals in a PSoC design. As an added bonus, it discusses the composition of a USB transaction, including the packets and error checking, which are very useful if you need to debug a USB design.

This application note includes three code examples that cover the following topics:

1. Interrupt, bulk, and isochronous transfers
2. Control transfers and vendor commands
3. Increasing throughput with DMA in PSoC 3

If you do not have any prior experience with USB, Cypress recommends that you read the application note AN57294 - USB 101: Introduction to Universal Serial Bus. For additional USB topics and examples, see the Related Resources section.

Application Note Requirements

This application note is intended for use with the following hardware/software and is required to fully evaluate the projects associated with this application note:

- PSoC Creator 2.0
- PSoC Programmer 3.13
- Cypress SuiteUSB 3.4.6
- PSoC 3 and PSoC 5 Development Kit (choose one)
  - CY8CKIT-001
  - CY8CKIT-030
  - CY8CKIT-050

PSoC Programmer and PSoC Creator are required to build/compile the USB projects and program the PSoC device. However, Cypress SuiteUSB is essential to test the example projects. SuiteUSB provides multiple graphical user interfaces (GUIs) that you can use to perform specific communication between the device and host, similar to what a custom GUI does. SuiteUSB also provides a vendor specific driver, CYUSB.sys, to test the various transfer types and vendor commands. For this application note, the drivers that will be used have been certified by Windows Hardware Quality Labs (WHQL) to provide a simplified driver install experience in Windows operating systems.

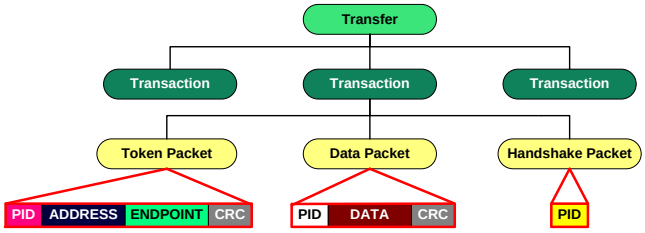
USB Transfers

Transfers are essential to exchange information between a USB device and a USB host. They enable communication between the device and the host. While most users only think of transfers as a high level aspect, there are many back and forth communications occurring continuously behind the scenes. The device and the host are constantly in communication with each other.

USB Transfer Structure

During the enumeration process of a USB device, one of the first things that the host does is request the device descriptor. The entire process of making the request for the device descriptor, receiving the device descriptor information, and host acknowledging successful reception of the data is the transfer. The transfer, however, comprises multiple stages called 'transactions'. Each transfer consists of one or more transactions and in the case of the device descriptor request, there are three. The first transaction is the Setup transaction, which is where the actual request is made to the device. The second transaction is the Data transaction, where the descriptor information is sent to the host. Finally, the third transaction is the handshake transaction where the host acknowledges receiving the packet. Then, you must go a level further. Each transaction is made up of multiple packets. Each transaction contains a token packet at minimum. Inclusion of a data packet and handshake packet can vary depending on the transfer type.

Figure 1. Structure of a USB Transfer



Notes

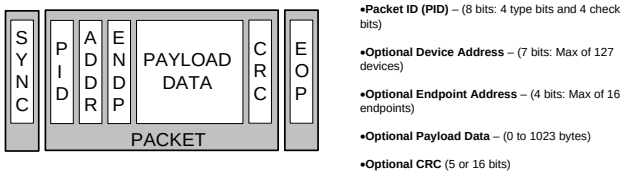
- Each transfer contains one or more transactions.
- Each transaction always contains a token packet. A data packet and handshake packet may be included depending on the transaction type.

Interrupt, bulk, and control transfers always include a token, data, and handshake packet with each transaction. Control transfers take this a step further. Remember that control transfers have three stages - Setup, Data, and Status. Each one of these stages contains a token, data, and handshake packet. Therefore, while an Interrupt and Bulk transfer have a minimum of three packets, a control transfer has nine or more with a data stage and six or more without a data stage.

Transfer Composition

A USB packet has a general template structure (see Figure 2). A total of five possible fields can be populated. Four of these fields are optional, while one is required. To get an idea of which fields are used in a packet, refer to Figure 2. For a detailed explanation of the composition of a packet, refer to the Communication Protocol section of AN57294.

Figure 2. USB Packet Contents



The Packet ID is the only required field in a packet. The Device Address, Endpoint Address, Payload Data, and CRC are filled depending on which packet type is sent. Packet IDs (PID) are the heart of a USB packet. As the name implies, it identifies the packet. There are different PIDs depending on which packet is sent (see Table 1).

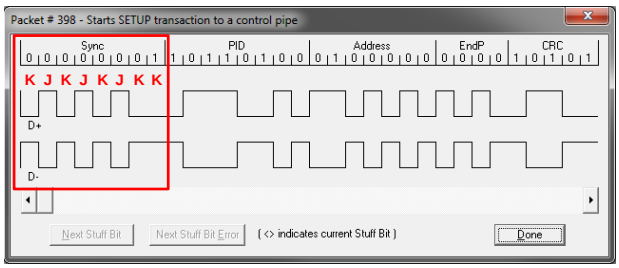
Table 1. Packet ID Information

Packet Type	PID Name	PID [3..0]	Description
Token	OUT	0001b	Address + endpoint number in host-to-function transaction.
	IN	1001b	Address + endpoint number in function-to-host transaction.
	SOF	0101b	Start-of-Frame marker and frame number.
	SETUP	1101b	Address + endpoint number in host-to-function transaction for SETUP to a control pipe.
Data	DATA0	0011b	Data packet PID even. Data Toggle
	DATA1	1011b	Data packet PID odd. Data Toggle
	DATA2	0111b	Data packet PID high-speed, high bandwidth isochronous Transaction in a microframe. **High Speed Only**
	MDATA	1111b	Data packet PID high-speed for split and high bandwidth isochronous transactions. **High Speed Only**
Handshake	ACK	0010b	Receiver accepts error-free data packet.
	NAK	1010b	Receiving device cannot accept data or transmitting device cannot send data.
	STALL	1110b	Endpoint is halted or a control pipe request is not supported.
	NYET	0110b	No response yet from receiver. **High Speed Only**
Special	PRE	1100b	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.

Packet Type	PID Name	PID [3..0]	Description
	ERR	1100b	(Handshake) Split Transaction Error Handshake (reuses PRE value). **High Speed Only**
	SPLIT	1000b	(Token) High-speed Split Transaction Token. **High Speed Only**
	PING	0100b	High-speed flow control probe for a bulk/control endpoint. **High Speed Only**
	Reserved	0000b	Reserved PID.

A packet transfer also includes a SYNC field and an End of Packet (EOP). The SYNC field is used for synchronization purposes and is included at the beginning of all packets. Because the receiver and sender of a packet do not share a clock, there must be a method to synchronize the incoming data with the local clock of the recipient of the packet. The SYNC field does this by allowing the recipient of the packet to adjust its clock before the packet arrives. Because the SYNC field is only used for synchronization purposes, it is not included or mentioned in most packet diagrams. On a full speed device, such as PSoC, the SYNC field is eight bits long, consisting of a sequence of KJKJKJKK states. Figure 3 shows an example. The SYNC field works in conjunction with bit stuffing to maintain clock accuracy.

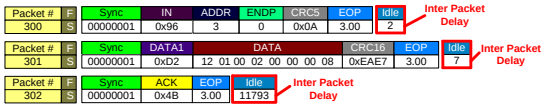
Figure 3. Full Speed SYNC Field



An End of Packet (EOP) signal is the conclusion of all packets. It returns the bus back to an idle state where it remains until the next SYNC occurs, followed by a packet. The EOP signal, similar to J and K states, differs on low, full, and high speed devices. On a full speed device, an EOP occurs when the data lines go into a SE0 state for 2-bit times, followed by a J state for 1-bit time.

An additional term you occasionally hear in relation to USB transactions is inter-packet delay. Inter-packet delay is the measured time (in bit times) between the EOP and the SYNC field. On a full speed device, the USB specification requires this to be a minimum of 2-bit times. This time enables a break in transmission for the device that sent the EOP to disable its output drivers.

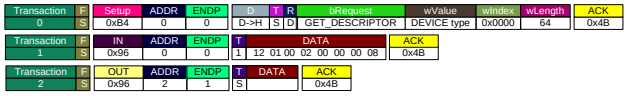
Figure 4. Bus Analyzer Showing Inter-Packet Delay



Here is an example to explain transfer composition in greater detail. AN57294 provides an example of an enumeration sequence in its appendix. An excerpt of that sequence is shown in Figure 5. This is the first step in communication with the device. Step 8 in AN57294 states the following:

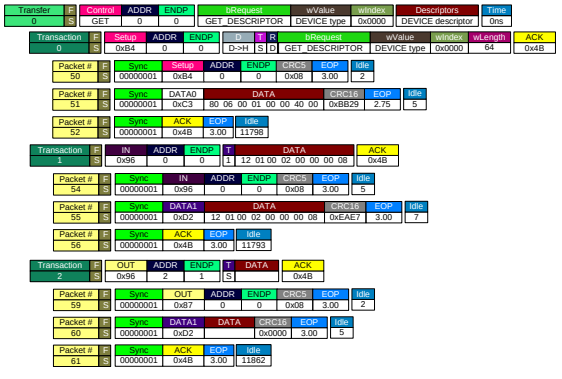
> *"The host will now begin its process of learning more information about the device starting with learning the maximum packet size of the default pipe (i.e. Endpoint 0). The host will start by issuing a GET_DESCRIPTOR request to the device. The device will begin to send the descriptors that were discussed in the USB Descriptor section of the application note. In the Device Descriptor, the 8th byte (bMaxPacketSize0) contains information regarding the max packet size for EP0. A Windows host will request 64-bytes, but after only receiving 8 bytes of the device descriptor, it will begin the status stage of the transfer and request the hub to reset the device again."*

Figure 5. GET_DESCRIPTOR Request during Enumeration



These three transactions are part of a transfer and each transaction has multiple packets associated with it (see Figure 6). You can see that the structure of this USB transfer follows the structure seen in Figure 1 on page 3.

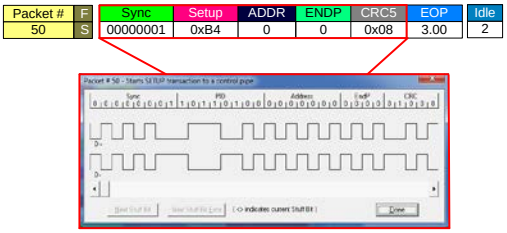
Figure 6. GET_DESCRIPTOR Request during Enumeration



In the GET_DESCRIPTOR example, this is a control transfer with a data stage. Transaction 0 is the Setup stage, Transaction 1 is the Data stage, and Transaction 2

is the Status stage. Note that a total of nine packets are needed to complete this transfer. Focusing on transaction 0, you can see that the token packet (Packet 50) contains the PID for a SETUP, the address of the device, the endpoint number, and the CRC. The data packet (Packet 51) has the PID for the Data Toggle, the data payload, and a CRC. Finally, the handshake packet (Packet 52) has a PID for a handshake for that transaction. The packet is the lowest level at which to look at USB communication with the exception of the actual D+ and D- logic levels as shown in Figure 7.

Figure 7. Composition of a USB Packet



USB Transfer Types

As mentioned earlier, packets can be transferred in many different transfer types. Each has its own advantages and disadvantages. Table 2 shows an overview of the features and capabilities of each of the transfer types. The following sections describe these transfer types in detail.

Table 2. Endpoint Transfer Type Features

Transfer Type	Control	Interrupt	Bulk	Isochronous
Typical Use	Device Initialization and Management	Mouse and Keyboard	Printer and Mass Storage	Streaming Audio and Video
Low Speed Support	Yes	Yes	No	No
Error Correction	Yes	Yes	Yes	No
Guaranteed Delivery Rate	No	No	No	Yes
Guaranteed Bandwidth	Yes (10%)	Yes (90%/80%)[1]	No	Yes (90%/80%)[1]
Guaranteed Latency	No	Yes	No	Yes
Maximum Transfer Size	64 Bytes	64 Bytes	64 Bytes	1023 Bytes
Maximum Transfer Speed	832 KB/s	1.216 MB/s	1.216 MB/s	1.023 MB/s

[1] Shared bandwidth between Isochronous and Interrupt.

Interrupt Transfers

When you initially think of an interrupt transfer, you may think that whenever data is ready to be sent or received, the host or device stops what it is doing to address the transaction. This is not what happens with an interrupt transfer. Interrupt transfers are regularly scheduled IN and OUT transactions that occur at an interval that is defined in the Endpoint Descriptor for a particular endpoint. At that defined interval, the host guarantees to perform the transaction at least that often. However, the transaction may occur more frequently than specified. Interrupt transfers are useful for applications where you want to send periodic status updates.

These transfers are also useful for sending small quantities of data within a specific amount of time, such as mice, keyboards, game controllers, and hub status reports. This transfer type is used in these devices because when you input something into a PC or a game console, you want to feel that the stimulus occurs instantly and that you do not experience any noticeable delay. Interrupt transfers provide this with their guaranteed latency.

The disadvantage of an interrupt transfer is the lack of a guaranteed delivery rate. If requests are being NAKed, the device retries at the next polling interval. If a design has a 10 ms interval, this can add significant delay. Therefore, the time it takes to deliver data is highly dependent on any issues that occur with the handshake.

The frequency of data delivery depends on the interval rate in the endpoint descriptor. This interval can vary from 1 to 255 ms on a full speed device like PSoC (low and high speed have other ranges). This value can be increased in 1 ms intervals.

Interrupt transfers are supported by all USB speeds. USB device support of Interrupt transfers, in a specific application, is optional. Although the USB specification does not require this transfer type to be supported, some USB classes do. The packets in an interrupt transfer can vary in size depending on the bus speed. While the data payload of a low speed Interrupt transfer can vary from 1 to 8 bytes, the data payload of a full speed transfer can vary from 1 to 64 bytes, and the data payload of a high speed transfer can vary from 1 to 1024 bytes.

Table 3.Capabilities of a Full Speed Interrupt Transfer

Data Payload (Bytes)	Frame Bandwidth per Transfer	Max Transfer per Frame	Max Transfer Speed
1	1%	107	107 KB/s
2	1%	100	200 KB/s
4	1%	88	352 KB/s
8	1%	71	568 KB/s
16	2%	51	816 KB/s
32	3%	33	1.056 MB/s
64	5%	19	1.216 MB/s
Protocol Overhead (13 Bytes)		3 SYNC bytes + 3 PID bytes + 2 Endpoint bytes + 2 CRC bytes + 3 byte inter-packet delay	

Figure 8 shows the flowchart with the various actions that can occur in an IN interrupt transaction. As mentioned earlier, the host periodically polls the interrupt endpoint. The endpoint is polled because the host has to initiate the transaction and it needs to check if data is ready to be transmitted. After the host sends the IN token, the device responds with a data packet if the device has data ready, a NAK if it does not have data ready and queued up, a Stall if there is an issue with the endpoint, or no response at all if there is something wrong with the request (such as the request becoming corrupt). Upon receiving the packet, the host will either ACK, letting the device know that it successfully received the data, or not send any response, indicating that the packet became corrupted during transmission. In that case, the device tries to resend the data upon the next polling interval.

Figure 9 shows the transaction structure of an interrupt (and bulk) IN transaction.

Figure 8. Flowchart of an IN Interrupt Transaction

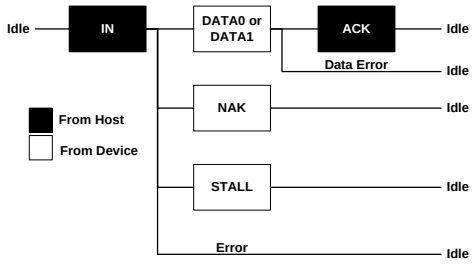
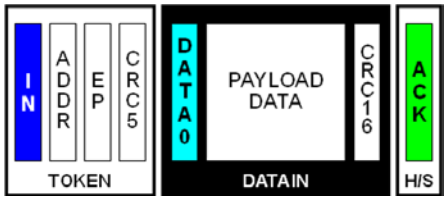


Figure 9. Structure of an IN Interrupt/Bulk Transaction



On the other side of things, when a host wants to send interrupt data to a device, it issues an OUT token followed by a data packet. If either of these packets becomes corrupted, then there is no response (no handshake) and the bus returns to an idle state. If the device endpoint buffer is empty and able to receive the data, then the device issues an ACK handshake indicating that it successfully received the data. If the endpoint buffer is full because of the endpoint buffer being full from a previous transaction, then an NAK handshake is returned. Similar to an IN transaction, if an error occurs at the endpoint, then a STALL handshake is issued. This is illustrated in the Figure 10. Additionally, the structure of an OUT interrupt/bulk transaction is shown in Figure 11.

Figure 10. Flow of an OUT Interrupt Transaction

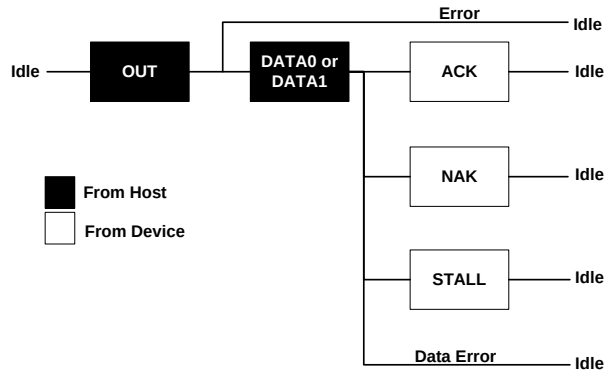
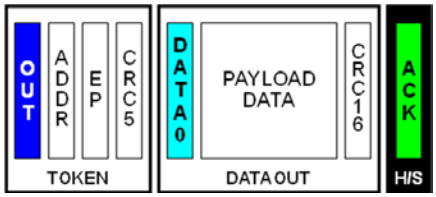


Figure 11. Structure of an OUT Interrupt/Bulk Transaction



Using Table 4 as a reference, Figure 12 and Figure 13 show an example of both an Interrupt IN and Interrupt OUT endpoint descriptor. Both these example descriptors are configured for a maximum packet size of 64 bytes (40h) and for an interval of 10 ms, denoted by 0Ah.

Table 4. Endpoint Descriptor

Offset	Field	Size(Bytes)	Description
0	bLength	1	Length of this descriptor = 7 bytes
1	bDescriptorType	1	Descriptor type = ENDPOINT (05h)
2	bEndpointAddress	1	Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction. Ignored for Control 0 = OUT endpoint 1 = IN endpoint
3	bmAttributes	1	Bits 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt If not an isochronous endpoint, bits 5..2 are reserved and must be set to zero. If isochronous, they are defined as follows: Bits 3..2: Synchronization Type 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4: Usage Type 00 = Data endpoint 01 = Feedback endpoint 10 = Implicit feedback Data endpoint 11 = Reserved
4	wMaxPacketSize	2	Maximum packet size for this endpoint
6	bInterval	1	Polling interval in milliseconds for interrupt endpoints (1 for isoch endpoints, ignored for control or bulk)

Figure 12. Example Endpoint Descriptor for IN Interrupt Endpoint

```
/*******************************************
Endpoint Descriptor
*******************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x81u,
/*  bmAttributes               */ 0x03u,
/*  wMaxPacketSize             */ 0x40u, 0x00u,
/*  bInterval                  */ 0x0Au,
```

Figure 13. Example Endpoint Descriptor for OUT Interrupt Endpoint

```
/*******************************************
Endpoint Descriptor
*******************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x02u,
/*  bmAttributes               */ 0x03u,
/*  wMaxPacketSize             */ 0x40u, 0x00u,
/*  bInterval                  */ 0x0Au,
```

Bulk Transfers

The primary use of Bulk transfers is to transfer large amounts of non-periodic burst data, such as writing a file to a USB flash drive, or sending a file to a printer. These transfers use any spare time (idle time) on the bus that is not used by Interrupt and Isochronous transfers. The more available the bus is, the faster the data is transferred until being limited by the bus speed. If the bus is tied up with a video stream from a webcam (which typically uses Isochronous), then the process of transmitting data with a bulk transfer can be much slower.

Bulk transfers are supported on full speed and high speed devices, but not on low speed devices. Similar to an Interrupt transfer, a device is not required to support bulk transfers. A specific device class may, however, require the support of bulk transfers. An example of a class that requires Bulk endpoint support is a Mass Storage Class device.

With interrupt transfers, you can declare a maximum packet size in the Endpoint Descriptor of up to certain values. For example, with a full speed device, you have a maximum packet size of anywhere between 1 and 64 bytes. With Bulk endpoints, the maximum packet size also depends on speed, but there is not as much flexibility in maximum packet size selection. On a full speed device, the maximum packet size must be 8, 16, 32, or 64 bytes long. On a high speed device, the maximum packet size can be 512 bytes long. Even with these limited required packet sizes, it is acceptable if the actual data payload falls short of the maximum packet size. The data payload does not need to be padded with zeros.

Because bulk transfers do not have priority on the bus, it should not be used in any application that has time sensitive data to transmit. This is due to the inability to accurately predict when the data is actually delivered. Similar to interrupt transfers, errors are detected and the transmission of the packet is attempted again. However, unlike interrupt transfers, which wait until the next interval, bulk transfers instantly retry assuming there is available bandwidth.

Table 5. Capabilities of a Full Speed Bulk Transfer

Data Payload (Bytes)	Frame Bandwidth per Transfer	Max Transfers per Frame	Max Bandwidth
1	1%	107	107 KB/s
2	1%	100	200 KB/s
4	1%	88	352 KB/s
8	1%	71	568 KB/s
16	2%	51	816 KB/s
32	3%	33	1.056 MB/s
64	5%	19	1.216 MB/s
Protocol Overhead (13 Bytes)		3 SYNC bytes + 3 PID bytes + 2 Endpoint bytes + 2 CRC bytes + 3 byte inter-packet delay	

The packet structure of an IN and OUT Bulk transaction is identical to an Interrupt transaction (see Figure 14 and Figure 15). The same applies to the flow of a bulk transaction with the exception of an additional handshake packet on high speed devices (NYET).

Figure 14 shows an example of a Bulk IN endpoint descriptor and Figure 15 shows a Bulk OUT endpoint. The maximum packet size in the endpoint descriptor is also configured for 64 bytes. The Interval field lists a value of 0. Remember that bulk transfers do not use the interval field because they lack a guaranteed latency and use any available bus bandwidth. Values listed in the bInterval field represent nothing.

Figure 14. Example Endpoint Descriptor for IN Bulk Endpoint

```
/*******************************************
Endpoint Descriptor
*******************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x83u,
/*  bmAttributes               */ 0x02u,
/*  wMaxPacketSize             */ 0x40u, 0x00u,
/*  bInterval                  */ 0x00u,
```

Figure 15. Example Endpoint Descriptor for Out Bulk Endpoint

```
/*******************************************
Endpoint Descriptor
*******************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x04u,
/*  bmAttributes               */ 0x02u,
/*  wMaxPacketSize             */ 0x40u, 0x00u,
/*  bInterval                  */ 0x00u,
```

Isochronous Transfers

Isochronous transfers are intended for streaming data to a host through a constant and real time stream of information of pre-negotiated bandwidth on the bus. This continuous stream guarantees on-time delivery of data. There are no bottlenecks besides the overall bus speed to impede the data transfer. With this benefit of guaranteed delivery comes a single disadvantage − there are no handshake packets or retrials to send corrupted data. Errors in the data are detected through the CRC in the data packet, but that is where it ends. Errors are only detected and not corrected. After an error is detected, the device throws away the corrupted data and waits for the next transfer. Because of this, when designing an application that uses isochronous transfers, you must ensure that the application can accept occasional losses of data.

Isochronous transfers are not ideal for transferring data files, such as in mass storage devices, because missing/lost data is unacceptable. Rather, isochronous transfers are ideal for audio and video transfers where an occasional loss of data is acceptable and in many cases, unnoticeable. Depending on the application, isochronous transfers can also be used for basic data streaming.

Isochronous transfers are only supported by full and high speed devices. Low speed devices are unable to implement isochronous transfers. Just as with Interrupt and Bulk transfers, devices are not required to support them by default, but a specific device class may require device support. Examples of device classes that do require isochronous transfers are audio and video.

The maximum packet size of an isochronous transfer varies from full speed and high speed devices, but only slightly. On a full speed device, an isochronous transfer is limited to 1023 bytes of data and the maximum packet size reported in the endpoint descriptor can range from 0 to 1023 bytes. A high speed device, however, increases the maximum packet size to 1024 bytes with a maximum packet size reported in the endpoint descriptor ranging from 0 to 1024 bytes. Additionally, the data is unable to transmit in a single packet, the host may complete the data transfer across multiple transactions as needed.

There are two additional design considerations with an Isochronous supported device:

- **Bus bandwidth**. Many USB devices with isochronous endpoints have multiple alternative interface configurations (also known as alternate settings). One interface configuration supports a large payload while another configuration has a much smaller payload using a smaller configured isochronous endpoint, or perhaps even bulk endpoints. It is not uncommon to see an isochronous device with four or more alternate settings to accommodate the available bus bandwidth. Additionally, the USB specification requires that a zero bandwidth interface exists on isochronous designs. If the isochronous device controls the bus when it does not need to, then this is a waste of bus bandwidth and can pose a bottleneck for other USB devices. Being able to adjust the bus bandwidth reservation of the device helps avoid this issue.
- **Three times buffering**. When dealing with isochronous transfers, the device developer will often want one buffer actively transmitting, another buffer loaded and ready to transmit, and a third buffer being actively loaded. This is not required but a good practice when streaming data from something like an ADC to maximize throughput.

Table 6. Capabilities of a Full Speed Isochronous Transfer

Data Payload (Bytes)	Frame Bandwidth per Transfer	Max Transfers per Frame	Max Bandwidth
1	1%	150	150 KB/s
2	1%	136	272 KB/s
4	1%	115	460 KB/s
8	1%	88	704 KB/s
16	2%	60	960 KB/s
32	3%	36	1.152 MB/s
64	5%	20	1.280 MB/s
128	9%	10	1.280 MB/s
256	18%	5	1.280 MB/s
512	36%	2	1.024 MB/s
1023	69%	1	1.023 MB/s
Protocol Overhead (9 Bytes)		2 SYNC bytes + 2 PID bytes + 2 Endpoint bytes + 2 CRC bytes + 1 byte inter-packet delay	

Figure 16 shows the format and possible flow of an isochronous transfer. Note that unlike the other diagrams (such as Figure 8), there is no handshake. In the instance of an isochronous IN transaction, the IN token is either received or becomes corrupt. The situation is the same for the data packet. Similar scenarios occur with an isochronous OUT transactions.

Figure 16. Flowchart of an IN Isochronous Transaction

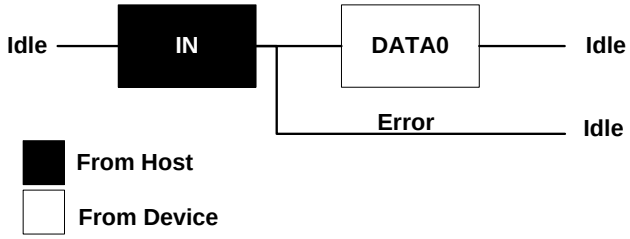
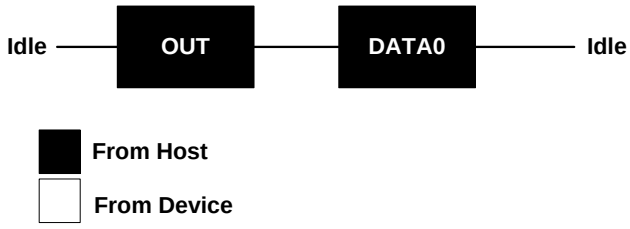


Figure 17. Flowchart of an OUT Isochronous Transaction



With isochronous transfers, a full-speed device or host controller can accept either DATA0 or DATA1 PIDs in data packets. However, a full-speed device or host controller must only send DATA0 PIDs in data packets. Because of this, full speed isochronous devices do not support data toggling. To implement error checking in isochronous transactions, you need to rely on checking for bit stuffing errors or CRC errors. Figure 18 and Figure 19 show block diagram examples of isochronous transactions, while Figure 20 shows an example of an Isochronous OUT transaction, captured by a bus analyzer. Note the unchanging Data Toggle bit and the lack of a handshake packet.

Figure 18. Example of an IN Isochronous Transaction

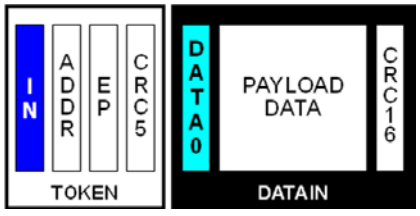


Figure 19. Example of an OUT Isochronous Transaction

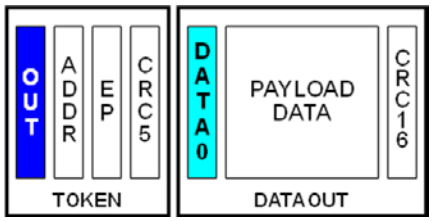


Figure 20. Isochronous 8-byte OUT Transactions on Bus Analyzer

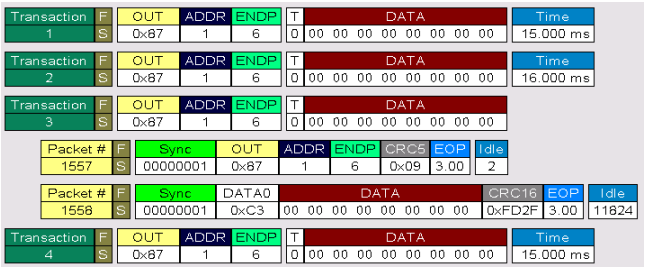


Figure 21 shows an example of an isochronous IN endpoint descriptor and Figure 22 shows an isochronous OUT descriptor. The maximum packet size in both endpoint descriptors is configured for 1023 bytes and the interval is configured to 1ms. The endpoint descriptor length and the descriptor type remain constant at 07h and 05h. Starting with bEndpointAddress, note that in Figure 24 bit 8 is set to '1', indicating an IN endpoint. Bits 3..0 have a value of 05h, indicating that this is endpoint number 5. Additionally, bmAttributes has a value of 01h. According to the endpoint descriptor template, this indicates that the endpoint is an isochronous endpoint. Because these are isochronous endpoints, bits 4..2 of bmAttributes now have significance. Bits 3..2, which set the Synchronization Type, are set to 00h indicating there is no synchronization. Also, bits 5..4 are set to 00h indicating that this endpoint is configured as a data endpoint.

Figure 21. Example Endpoint Descriptor for IN Isochronous EP

```
/*********************************************
Endpoint Descriptor
*********************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x85u,
/*  bmAttributes               */ 0x01u,
/*  wMaxPacketSize             */ 0xFFu, 0x03u,
/*  bInterval                  */ 0x01u,
```

Figure 22. Example Endpoint Descriptor for OUT Isochronous EP

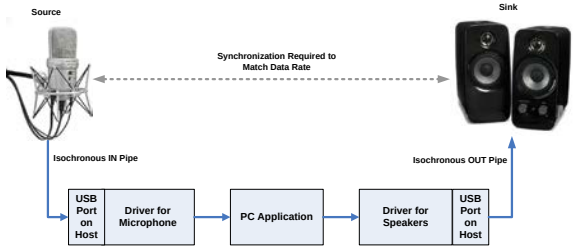
```
/*********************************************
Endpoint Descriptor
*********************************************/
/*  Endpoint Descriptor Length */ 0x07u,
/*  DescriptorType: ENDPOINT   */ 0x05u,
/*  bEndpointAddress           */ 0x06u,
/*  bmAttributes               */ 0x01u,
/*  wMaxPacketSize             */ 0xFFu, 0x03u,
/*  bInterval                  */ 0x01u,
```

Synchronization of Isochronous Transfers:

Isochronous transfers are interesting because they have multiple synchronization options for use when transmitting between two isochronous devices, rather than between a device and a host. It is one thing to transmit data as fast as the device can, but it is another thing to have a device that is ready to do something with that data when it arrives. Because of the continuous stream of data, it is important to make sure bottlenecks, such as two devices not able to keep up with one another, are eliminated. You commonly see these synchronization options used on audio class devices. If you want to stream isochronous data to a host, without a destination device, you typically use 'No Synchronization' for the synchronization type and Data Endpoint for the Usage Type in the Endpoint descriptor.

Isochronous transfers have the ability to synchronize to a data source, a recipient, or the bus's start of frame packets. With these synchronization options comes new terminology: Source and Sink. An Isochronous source device is a device that data you want to stream is being collected. For example, if you have a USB microphone with an ADC that is sampling voice data and streaming it over an isochronous IN endpoint, that is a source. On the other end, an isochronous sink is a device that is receiving data from the Source and doing something with that received data. An example is a pair of USB speakers that receive the digitalized voice data from the source microphone example, and convert that digital signal into sound waves.

Figure 23. Use Case Example of Isochronous Synchronization



There are three synchronization options available for isochronous source and sink devices: Asynchronous, Synchronous, and Adaptive.

- **Asynchronous**: Devices that use this synchronization method do not share a common clock source. They use an internal free running or external clock source to generate their sample rate. They are not permitted to synchronize to the SOF or any other USB clock. Because of this lack of common clock to provide a synchronized sample rate, the sample rate of the two

devices are not matched. Additionally, the devices do not have a way to adjust the rate at which they handle data. To compensate, the burden needs to be placed on the host application to provide data rate matching between the two devices. To do this, an Asynchronous sink device provides feedback regarding the data rate at which it operates. The source provides feed forward information to the host with regard to its sample rate. This information is provided implicitly by the number of samples produced for each frame. By knowing the data rate of the source, and the data rate handling capabilities of the sink, the host application can buffer the data between the devices to help synchronize. This adds overhead to the implementation. Asynchronous devices operate at a fixed or limited number of data rates.

The USB specification gives an example of an asynchronous source, such as an audio CD player, that provides its data on an internal clock or resonator. Because of that fixed clock, the CD player has a fixed sample rate for a CD, 44.1 kHz in this case. In this case, you have a fixed sample rate that is independent of the USB. An asynchronous sink is a pair of speakers that run off their own internal clock.

- **Synchronous:** Devices that use this synchronization method share the SOF as a common clock source between them. The source and sink devices must generate their sample clock through the 1 ms SOF with the use of a programmable PLL. Because both devices reference the same clock, there is no need for data rate matching by the host application. Synchronous devices operate at a fixed or limited number of data rates.

  The USB specification gives an example of a synchronous source as digital microphone that creates its sample clock from the SOF and produces a fixed number of audio samples each frame. Additionally, a synchronous sink, such as a pair of USB speakers, derives its sample clock from the SOF and consumes a fixed number of samples every frame.

- **Adaptive:** Devices that use this synchronization method have the greatest flexibility. While the other two synchronization methods require a fixed or limited number of data rates, Adaptive synchronization enables sink and source devices to operate at any data rate that is within their operating range. Adaptive source devices produce data at a rate that is controlled by the sink device. The sink provides feedback of its desired data rate. The source sample rate is then adjusted to match the requested rate. The sink device detects the data rate based on the number of samples it receives within a certain amount of time. If the received rate is lower or higher than desired by the sink, it feeds that information back to the host.

  The USB specification gives an example of an adaptive source as a CD player that contains an adaptive Sample Rate Converter (SRC), so the output sample frequency does not need to be 44.1 kHz like the standard audio CD player in the asynchronous example, but can be any other frequency within the operating range of the CD players SRC. An Adaptive sink example includes digital speakers, headphones, and so on.

Table 7. Synchronization Characteristics

	Source	Sink
Asynchronous	Free running sample rate. Provides implicit feedforward.	Free running sample rate. Provides explicit feedback.
Synchronous	Sample rate locked to SOF. Uses implicit feedback.	Sample rate locked to SOF. Uses implicit feedback.
Adaptive	Sample rate locked to sink Uses explicit feedback.	Sample rate locked to source. Uses implicit feedforawrd.

To maintain a constant and uninterrupted flow of data from device A to device B, some type of synchronization feedback needs to be implemented to ensure the data rate on the source matches the data rate on the sink. The USB specification defines two ways in which information regarding the data rate can be communicated: explicit and implicit feedback. With explicit feedback, a dedicated isochronous pipe is defined in the endpoint descriptor and used to provide feedback information regarding the data rate. With implicit feedback, the data rate information is extracted from the data stream. In the case of Synchronous synchronization, rate information is derived from the Start of Frame.

Feedback and Feedforward

The exchange of this information can also occur in two different fashions: Feedback and Feedforward. With feedback information, the information regarding the data rate is sent in the direction opposite to the data flow. With Feedforward, information is sent in the direction of the data flow.

In Table 7, note that only Adaptive sources and Asynchronous sinks provide and use explicit feedback information. Thus, configuration for explicit feedback is only required when using that combination of synchronization. Implementing explicit feedback requires configuring a feedback endpoint in an isochronous design. By referencing the Endpoint Descriptor Table in

Table 4, you can see that this is accomplished by setting the “Usage Type” in bmAttribues to 01b, defining it as a Feedback endpoint. The feedback information sent across this endpoint consists of a 3-byte data packet. This packet contains information that represents the average number of samples a device must produce or consume for each frame in order to produce the desired sampling frequency within 1 Hz. The disadvantage of using this method is the overhead in comprising the information in the data packet with the required information outlined in Section 5.12.4.2 of the USB Specification. Additional information can be found in the USB Audio Class Specification.

Implicit feedback is much simpler to implement and can be implemented on all other combinations of isochronous synchronization besides the Adaptive source and Asynchronous sink. As mentioned earlier, implicit feedback uses information extracted from the data stream or derived from the SOF. Configuring an endpoint to use this type of feedback requires setting the “Usage Type” in bmAttribues to 10b, defining it as an Implicit Feedback Endpoint. Because the data rate is derived from already existing information, such as the data stream and SOF, no additional complexity is added.

Control Transfers

Control transfers are used to identify, configure, and control USB devices, rather than stream data. Control transfers are how the host gathers information regarding a USB device and how the host is able to configure the device. A Control Transfer always occurs through Endpoint 0 (EP0), which is also why EP0 is referred to as the control endpoint. A control transfer consists of three stages. These stages are a Setup Stage, a Data Stage, and a Status Stage.

Table 8. Capabilities of a Full Speed Control Transfer

Data Payload (Bytes)	Max Transfers per Frame	Frame Bandwidth per Transfer	Max Bandwidth
1	32	3%	32 KB/s
2	31	3%	62 KB/s
4	30	3%	120 KB/s
8	28	4%	224 KB/s
16	24	4%	384 KB/s
32	19	5%	608 KB/s
64	13	7%	832 KB/s
Protocol Overhead (45 Bytes)		9 SYNC bytes + 9 PID bytes + 6 Endpoint bytes + 6 CRC bytes + 8 Setup data bytes + 7 byte inter-packet delay	

There are three types of control transfers: Control Read, Control Write, and Control with No Data. Choosing the transfer depends on whether receiving data from the device or writing data to it is required. In many USB requests, all the required information can usually be transferred in the data packet of the Setup stage. Figure 24 and Figure 25 show the flow of a control write and a control read transfer. If you look at data and status stages, note that they are the same as a bulk and interrupt transfer. The Setup stage is, however, different. It has a handshake phase but there is only one acceptable response, which is an ACK. You cannot NAK or Stall a Setup stage. In the instance of no response, the packet is resent.

Figure 24. Control Write Transfer

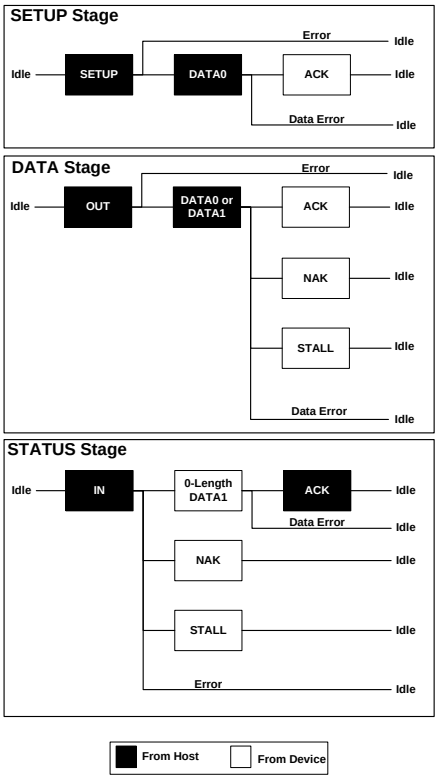
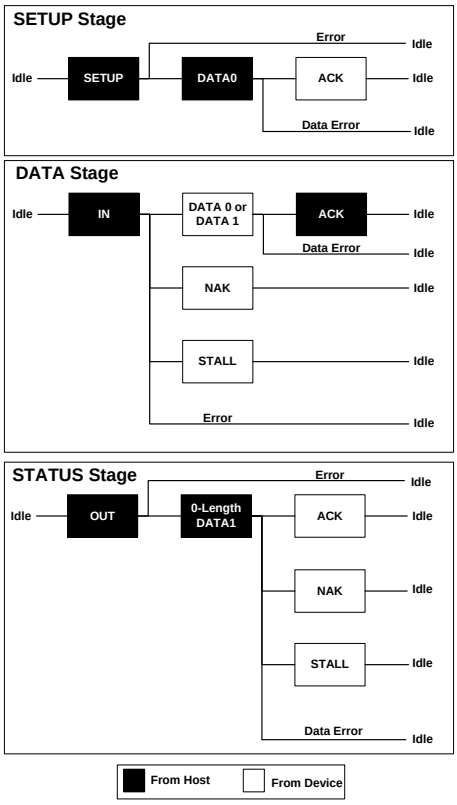
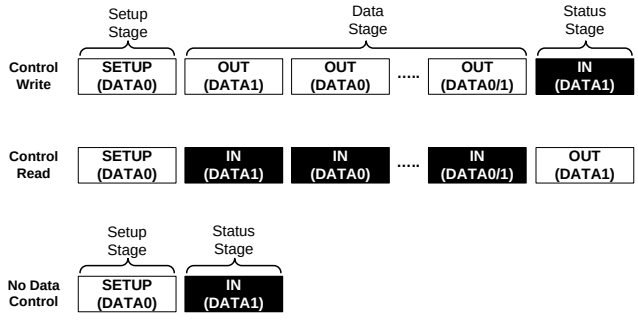


Figure 25. Control Read Transfer



In Figure 24 and Figure 25, note the data toggle. The Setup stage always has a constant DATA0, the Status stage always has a constant DATA1, and the Data stage alternates between DATA0 and DATA1, always beginning with DATA1. A better depiction of this is illustrated in Figure 26.

Figure 26. Possible Scenarios of a Control Transfer



Control write transfers return information regarding the transaction in the data packet of the status stage. Control read transfers return information regarding the transaction in the handshake packet of the status stage.

Table 9. Possible Status Stage Responses

Status Response	Control Write	Control Read
Request Complete	Zero-length Data Packet	ACK Handshake
Request Failed	STALL Handshake	STALL Handshake
Device Busy	NAK Handshake	NAK Handshake

Similar to bulk transfers, the data size is dependent on device speed and is limited to a maximum of 8 bytes for low speed, maximum of 8, 16, 32, or 64 bytes on full speed, or a maximum of 64 bytes on high speed. Data packets in control transfers are allowed to be less than the maximum packet size, specified in the Device Descriptor. The data stage is completed upon either transferring the exact amount of data specified in the status stage, transfers a data payload less than what is specified in wMaxPacketSize, or a zero length packet is received. When this occurs, the host moves on to the status stage.

Because of the high overhead of a control transfer, 45 bytes versus 13 bytes of an interrupt transfer, this is not an efficient way to constantly transfer data despite their guaranteed bandwidth, error correction, and lack of intervals. If control transfers need to exceed the 10% bandwidth, and there is spare bandwidth on the bus, they may do so. Additionally, if they go under the 10% reserved bandwidth, they may reallocate bandwidth to bulk transfers.

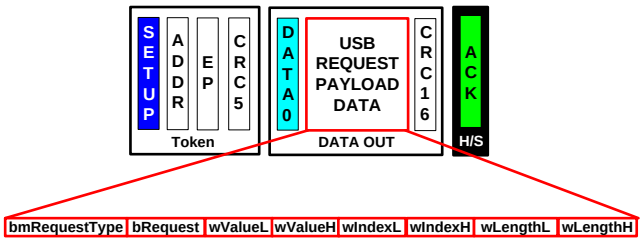
USB Requests

USB requests are a major part of USB control transfers. The USB Specification describes three types of requests to communicate to a device: standard, class, and vendor specific. All USB devices must support standard requests, which are basic status and configuration commands. Class and vendor-specific commands, however, are optional. Many USB devices are under a predefined USB class specification and support class commands. Examples of the more common device specifications are Human Interface Device (HID), Mass Storage Device, or Communication Device. There are some devices that do not fall under these or any other predefined classes. These are called Vendor-Specific devices and are common with USB devices as it allows device designers to create custom communication protocols for USB.

USB Requests are how the USB host commands the device to provide it with information or perform specific control and configuration operations. These requests are sent in the 8-byte data packet of the Setup phase of a Control transfer. As mentioned in the Control Transfer

section, these requests come across Endpoint 0. If we look at a transfer of a USB request, you can see something similar to Figure 27.

Figure 27. Control Transfer of USB Request



The Token Packet (SETUP) identifies the receiver and identifies the transaction as a Setup transaction. The Data Packet (DATA0) transmits the request and any related information to the request. The Handshake Packet (ACK) identifies the devices acknowledgement of the transaction.

In Figure 27, note the various parameters such as bmRequestType, bRequest, wValue, and so on. These parameters are what comprises the 8 bytes of the data in the Setup stage that the firmware decodes to know which request was issued and how to handle it. The format of a USB request is illustrated in Table 10.

Table 10. USB Request Format

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bitmap	Characteristics of request (refer to Table 11)
1	bRequest	1	Value	Specific request (refer to Table 12)
2	wValue	2	Value	Word-sized field that varies according to request
4	wIndex	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

- ***bmRequestType*** is a byte that defines the direction of the transfer, the type of request, and the recipient of the request. Bit 7 defines the transfer direction. Here, it is determined if the data is transferred from host to device, or device to host. Bit 6 and bit 5 define the request type of the transfer such as Standard, Class, or Vendor. Bits 4 though 0 define who the request is directed to. This can be a device, interface, endpoint, or other. An example of this can be seen in Table 11.

Table 11. bmRequestType Composition

Bits	Field	Values
7	Direction	0: Host to Device (OUT) 1: Device to Host (IN)
6..5	Type	0: Standard 1: Class 2: Vendor 3: Reserved
4..0	Recipient	0: Device 1: Interface 2: Endpoint 3: Other 4-31: Reserved

- ***bRequest*** is a byte that specifies a request. Each request has a unique bRequest value. As mentioned earlier, all USB devices must respond to a Standard request. For simplicity, we use a standard request as an example. The USB Specification currently defines eleven standard requests for control transfers, which are listed in Table 12. Using Table 12 as a reference, we can see the various request numbers for each bRequest with regards to a standard request.
- ***wValue*** is a two byte value, which the host uses to pass information to the device. What specifically goes into these two bytes is dependent on what request has been made.
- **wIndex** is a two byte value that the host uses to pass additional information to the device. Normally, this value references an interface or endpoint. What specifically goes into these two bytes depends on the request that is made.
- **wLength** is a two byte value that contains the number of bytes in the data stage that will follow the SETUP stage.

If the direction of the transaction, indicated by the Direction bit in bmRequestType, is an input request (device-to-host), the device can never send more than the number of bytes indicated by wLength. It may, however, return less. If the request is an output (host-to-device), then wLength indicates the exact number of bytes to follow. A value 0 in wLength indicates that this is not a data stage.

Now, let us look at an example. As mentioned, there are 11 standard requests for USB that must be supported. Table 12 lists those requests and the parameters associated with them.

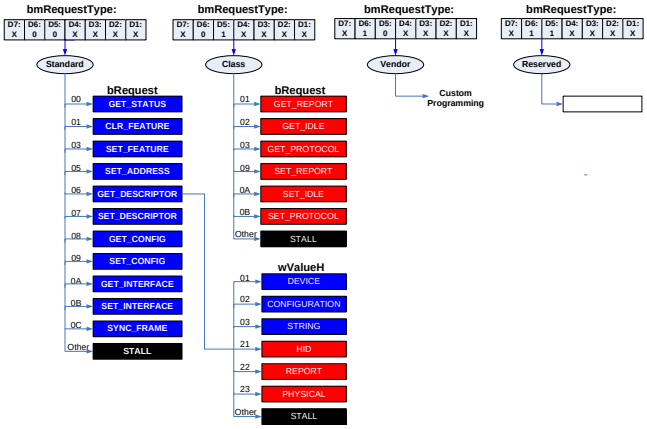
Table 12. Standard Device Request Table

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000b 00000001b 00000010b	CLEAR_FEATURE (01h)	Feature Selector	Zero Interface Endpoint	Zero	None
10000000b	GET_CONFIGURATION (08h)	Zero	Zero	One	Configuration Value
10000000b	GET_DESCRIPTOR (06h)	Descriptor Type and Index	Zero or Language ID	Descriptor Length	Descriptor
10000001b	GET_INTERFACE (0Ah)	Zero	Interface	One	Alternate Interface
10000000b 10000001b 10000010b	GET_STATUS (00h)	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000b	SET_ADDRESS (05h)	Device Address	Zero	Zero	None
00000000b	SET_CONFIGURATION (09h)	Configuration Value	Zero	Zero	None
00000000b	SET_DESCRIPTOR (07h)	Descriptor Type and Index	Device Address	Descriptor Length	Descriptor
00000000b 00000001b 00000010b	SET_FEATURE (03h)	Feature Selector	Feature Selector	Zero	None
00000001b	SET_INTERFACE (0Bh)	Alternative Setting	Alternate Setting	Zero	None
10000010b	SYNCH_FRAME (0Ch)	Zero	Zero	Two	Frame Number

For example, if a SET_ADDRESS request was sent, you may expect the bmRequestType to be configured with Host-to-Device as the direction, Standard as the type, and the Device as the recipient. According to Table 11, this gives us a value of 00000000b. We can confirm this is the case in Table 12. Additionally in Table 12, bRequest contains a value of 05h, wValue contains the device address, and all other fields (wIndex and wLength) are set to zero. There is no data phase associated with this request since we see zero in the wLength field.

Remember that Table 12 contains only standard requests. Class requests can be found in their supporting class documentation. For example, the HID class specification lists various class requests such as GET_REPORT and GET_PROTOCOL. Be sure to reference those documents when appropriate. Vendor requests, on the other hand, are not listed in any documentation since they are custom commands that you, the developer, create. When developing a vendor request, bmRequestType always follows the format defined in Table 11. However, you can define your own RequestType value, and populate wValue, wIndex, and Data Phase (Data phase only if wLength is greater than zero) as appropriate. Figure 28 shows the different paths bmRequestType can follow for various USB Requests.

Figure 28. USB Request Types Chart



By adjusting the bmRequestType field, you can issue the same request to different recipients. Table 13 shows an example of how we can issue the same request by simply changing the recipient (bits 4..0) in bmRequest Type

Table 13. Same Request with Different Recipients

Dir	Type	Recipient	Request	Result
IN	STD	DEV	GET_STATUS	Return device status to host
IN	STD	IF	GET_STATUS	Return interface status to host
IN	STD	EP	GET_STATUS	Return endpoint status to host

In some cases, the same request can be issued but with different request types. For example, there is a GET_DESCRIPTOR command under Standard Requests and one under HID Class Requests (with Standard and HID being the request types). The information that is returned depends on which request type is used. When bmRequestType = 00, bRequest specifies that a standard USB request is issued. When bmRequestType = 01, bRequest specifies that a device class request is issued. An example of this can be seen in Table 14.

Table 14. Same Request with Different Types

Dir	Type	Recipient	Request	Result
IN	STD	DEV	GET_DESCRIPTOR	Return DEVICE, CONFIG or STRING descriptor based on wValue
IN	CLS	DEV	GET_DESCRIPTOR	For HID class, return HID, REPORT or PHYSICAL descriptor based on wValue. Differs for other classes.

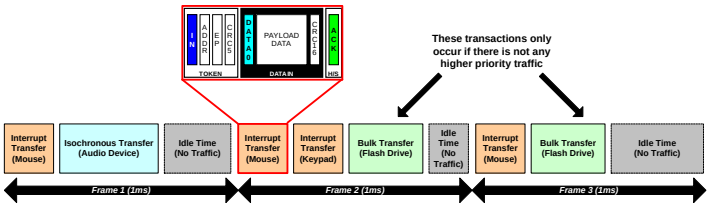
PSoC Creator automatically generates code to handle standard USB requests and some class requests. You as the developer only need to focus on loading data into and out of the endpoint buffers using the provided PSoC Creator APIs, which are described in the USBFS component datasheet. This application note discusses how to use those APIs in conjunction with custom functions to handle vendor specific commands later on in the code examples.

Bus Bandwidth Management

Any transfer performed by a USB device requires some fraction of the USB bandwidth. Since the host is responsible for controlling the transfers on the bus, the host is also responsible for managing the bandwidth on the bus. The process of assigning bandwidth on the bus is known as Transfer Management. It is not just the host that makes the decision. There are multiple components that work together to move data over the bus.

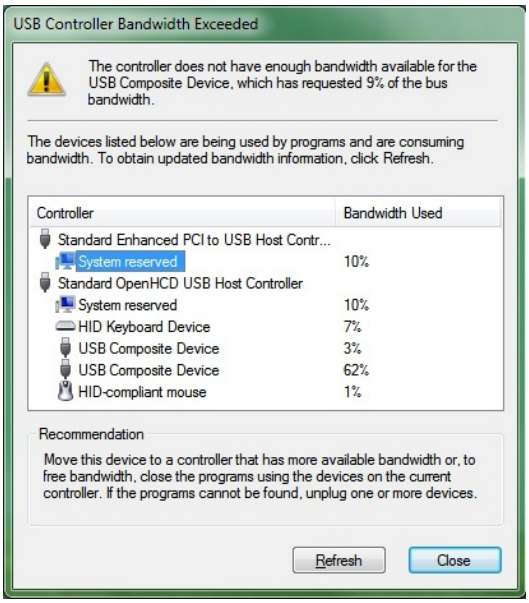
During the enumeration of a USB device, the host allocates bus bandwidth for Interrupt and Isochronous endpoints, and continues to do so as long as they remain connected to the bus. Bulk endpoints are not included since they do not have guaranteed bus bandwidth. Instead, they use whatever bandwidth is available. The USB specification places a limit on the allocation of bandwidth for these endpoints by stating that not more than 90% of a frame can be allocated to periodic transfers, which are interrupt and isochronous transfers, on a full speed device. This requirement is even stricter on a high speed bus, which regulates that no more than 80% of a microframe can be allocated for periodic transfers. On a full speed bus, this is not to say that the remaining 10% of the bus bandwidth is used for Bulk transfer, but this is the 10% reserved bandwidth for Control transfers. Bulk transfers only get what remains in USB frames from the interrupt or Isochronous transfers.

Figure 29. USB Traffic on the Bus Example



When a bus is overused, in a Windows environment, you can expect to see a window such as the one shown in Figure 30 appear. In this figure we can see two host controllers. In AN57294 - USB 101, it is discussed that there are often two host controller chips in a host. An EHCI controller for High Speed transfers and either an OHCI or UHCI for Low and Full Speed transfers. In the following figure, this machine has EHCI and OHCI controllers. Note that the various percentage of used bus bandwidth adding up.

Figure 30. USB Controller Bandwidth Exceeded



USB Error Correction and Detection

Error correction is a major part of USB communication. You need to be aware when something has gone wrong in your communication so that you can compensate or ignore data. There are two forms of error correction in a USB system. Error checking bits are included with packets and are calculated with a mathematical equation defined by the USB specification. This error checking is called the Cycle Redundancy Check (CRC). The second form of error correction is the data toggle. The data toggle is a form of synchronization between the device and host using the PID in a data packet. Keeping track of this data toggle can alert you to an error in the transaction.

Additionally, handshake packets are a form of error correction, though they are, by definition, not. However, handshake packets can give a lot of insight to the activity on the bus. How the device or host responds to something can, especially if it is a negative response, can indicate that is something is wrong.

CRC Check

CRC is commonly used by hard drives and communication interfaces such as CAN. A CRC is calculated for all token, data, and start of frame packets. The token and start of frame packets have a 5-bit CRC and the data packets have a 16-bit CRC.

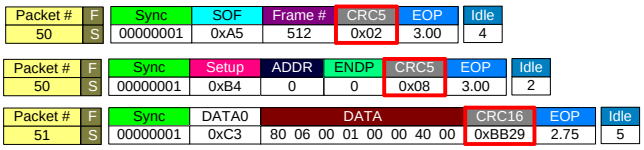
Table 15. CRC Reference Table

Packet	Fields Used in CRC	Number of CRC Bits
Start of Frame	Frame Number	5 bits
IN	Device Address and Endpoint Number	5 bits
OUT	Device Address and Endpoint Number	5 bits
SETUP	Device Address and Endpoint Number	5 bits
DATA0/DATA 1	Data Payload	16 bits

These CRCs are based on a generator polynomial. Details regarding the algorithm can be found in Section 3.5 of the USB 2.0 Specification. The CRC calculations are handled by the USB hardware. Because no software is required to handle the calculations, this application note does not discuss the math in detail.

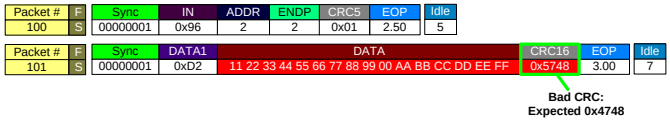
When a packet is transmitted, the device that does the transmission performs the calculation. The result of the calculation is then included with the packet transfer (see Figure 34).

Figure 31. CRCs in USB Packets



The recipient of the packet then performs the same calculation of what it received and compares the CRC that was transmitted to the CRC it calculated. If the two match, then the data is received without error and the ACK handshake is returned by the recipient. If the two CRCs do not match, then the recipient of the data never responds with a handshake, indicating there is something wrong with the transaction and causing the sender to resend the data. The resend occurs after a timeout period, which is ~16 bit times long. The only exception to this is with an isochronous transfer.

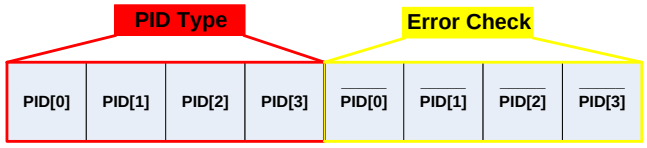
Figure 32. CRCs Error



PID Check

While token packets use a CRC against the device address and endpoint number, let us see how error checks are done against the PID. They implement something significantly simpler then a polynomial equation. 1's compliment is used to determine if there is an error in the PID. Figure 33 shows the composition of the PID. The receiver of the PID compliments the Error check bits and compares them to the PID Type bits. If they do not match, then the packet is ignored. There is not a retry event.

Figure 33. PID Check Format

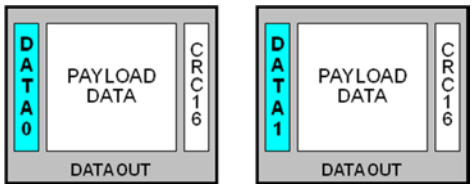


In the Transfer Composition section, Table 1 listed the various PID values. Note the table only lists 4 bits of an 8-bit value. The other four bits (MSbs) are the error checking.

Data Toggle

Data toggle ensures that data does not go missing when you have multiple data transactions. It acts as an error checking system. The idea behind the data toggle is to have an endpoint alternate between DATA1 and DATA0. The device and host keep track of the data toggle on a particular endpoint. When a USB device is configured, the data toggle is initially set to DATA0. This provides synchronization between the host and device by putting them on the "same page".

Figure 34. Data Toggle on Data Packets



When data is transferred, the receiver of the data compares the data toggle of the data packet with its own known data toggle. If these values match, then the receiver of the data issues an ACK as its handshake.

When the sender of the data packet receives this ACK, then it changes its Data Toggle value in preparation of the next transaction. If there is another data packet to transfer, then the Data Toggle changes to DATA1. This continues back and forth until all data has been transferred. An example of data toggle being used in USB transaction can be seen in Figure 35. This process is shown in Figure 36.

Figure 35. USB Bus Analyzer Trance of Data Toggle

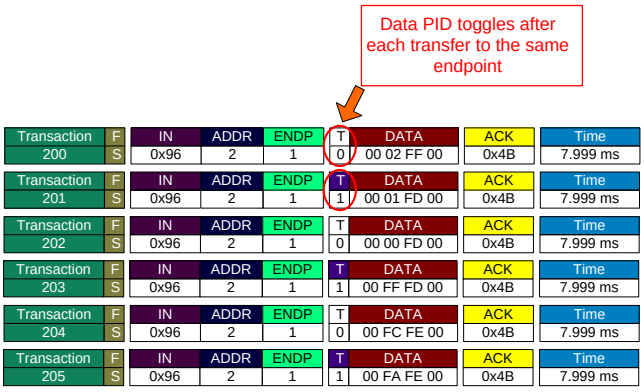
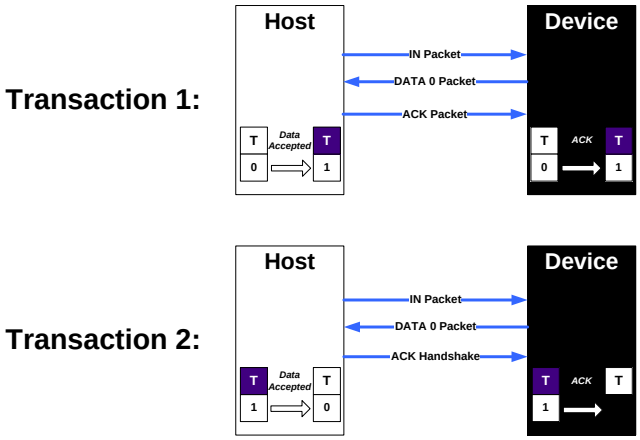
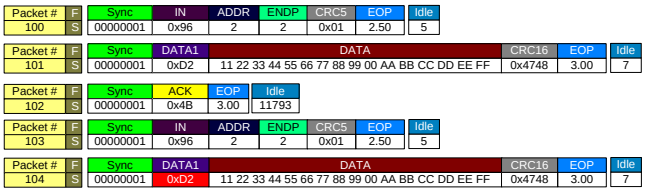


Figure 36. Example of Successful Data Toggles



However, if the receiver detects a mismatch between its data toggle and the data toggle of the packet, it responds by not sending a handshake packet.

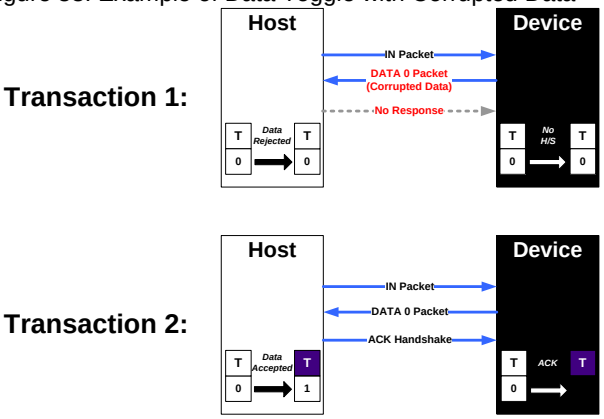
Figure 37. Data Toggle Error



There are two potential scenarios where you see the data toggle being extremely useful in error correction:

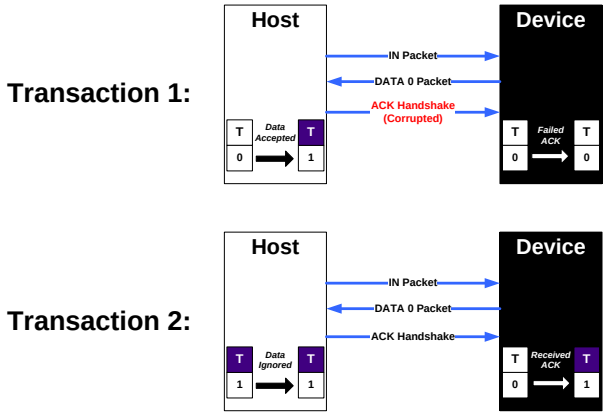
1. The receiver returns a NAK because it is not ready to send data or the receiver detects corrupted data (CRC Failure) and does not send a handshake. In these cases, the sender does not update its data toggle and attempts to resend the packet with the same data and data toggle. An example is seen in Figure 38.

Figure 38. Example of Data Toggle with Corrupted Data



2. If the receiver returns an ACK handshake and it becomes corrupted. In this case, the sender of the data assumes that the receiver did not get the data and attempts to resend the same data with the same data toggle. The receiver then does not update its data toggle, ignores the data, and returns an ACK handshake (see Figure 39).

Figure 39. Example of Data Toggle with Corrupted Handshake



Handshake

Handshake packets are essential in USB communication to ensure that data is properly received. They provide a closed loop system and contribute heavily to error correction and debugging in USB applications. In an ideal world, all USB transactions occur flawlessly and data does not become corrupt, there is always data ready to transmit, and all USB requests are understood. Since there is no such thing as an ideal USB transaction, we have handshake packets. These packets give us a closed loop system with Control, Interrupt, and Bulk transfers to provide two way communications between the device and the host.

AN57294 mentions that Handshake packets are used to conclude each transaction. Each handshake includes an 8-bit packet ID and is sent by the receiver of the transaction. There are multiple options for a handshake response and which ones are supported depend on the USB speed. Additionally, there are four standard handshake packets: ACK, NAK, STALL, and NYET. Since NYET is a high speed handshake, and PSoC is a full speed device, this application does not discuss it.

Handshakes are used to report many things about a device such as successful completion of data, request command acceptance or rejection, or flow control. Following are the three full speed handshakes:

- **ACK:** Acknowledge successful completion and error free receipt of a data packet. An ACK indicates that the packet was received without any data toggle, bit stuff, or CRC errors. An ACK can be issued by the host in the case of IN transactions and by the device in the case of OUT and SETUP transactions. Figure 40 shows an example of an ACK transaction on a bus analyzer.

Figure 40. Example of a NAK in a Transfer

Packet #	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
300	S	00000001	0x96	3	0	0x0A	3.00	2

Packet #	F	Sync	DATA1	DATA	CRC16	EOP	Idle
301	S	00000001	0xD2	12 01 00 02 00 00 00 08	0xEAE7	3.00	7

Packet #	F	Sync	ACK	EOP	Idle
302	S	00000001	0x4B	3.00	11793

- **NAK:** Negative acknowledgement. Indicates that a device was unable to accept data from the host in the case of a OUT request, or that the function does not have any data to transmit to the host in the case of an IN request. NAKs only occur in the data phase of an IN transaction or the handshake phase of an OUT transaction. Additionally, only a device can issue a NAK. The host is not capable of sending a NAK. They are used for flow control purposes to inform the host that the device is temporarily unable to transmit or receive data. Figure 41 shows an example of a NAK transaction on a bus analyzer.

Figure 41. Example of a NAK in a Transfer

Packet #	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
200	S	00000001	0x96	2	1	0x18	2.50	8

Packet #	F	Sync	NAK	EOP	Idle
201	S	00000001	0x5A	3.00	54

- **STALL:** Error indication sent by a device. Stalls are returned by a device in response to an IN token packet or an OUT data packet. Stalls indicate that the device is unable to transmit or receive data, or a request made through the control pipe is not supported. Stall handshake packets can only be sent by the device. A host cannot issue a stall. Figure 42 shows an example of a STALL transaction on a bus analyzer.

Figure 42. Example of a STALL in a Transfer

Packet #	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle
100	S	00000001	0x96	5	0	0x0B	2.50	8

Packet #	F	Sync	STALL	EOP	Idle
101	S	00000001	0x78	3.00	11883

Error Detection in PSoC 3 and PSoC 5

Detecting these errors on PSoC 3 and PSoC 5 is accomplished with the help of the SIE on the device. Located in the register reference of the PSoC 3/PSoC 5 Technical Reference Manual (TRM), there are multiple registers called non control endpoint control registers. These are the control registers for EP1 through EP8. Specifically, the register is named USB_SIE_EPx_CR0, where ‘x’ is replaced by an endpoint number. Looking at USB_SIE_EP1_CR0, for example, bit 6 is named err_in_txn.

Table 16. USB_SIE_EPx_CR0 Register Definition

Bits	7	6	5	4	3	2	1	0
Name	stall	err_in_txn	nak_int_en	acked_txn	mode			

The err_in_txn bit is set whenever an error is detected. For an IN transaction, this indicates that no response was received from the host. For an OUT transaction, this bit represents a receive error and includes PID error, CRC errors, and bit stuff errors. Users who wish to implement error checking should read this register after polling the OUT endpoint buffer to see if it is full. If this bit is set, then disregard the data. Table 17 provides a list of the different CR0 registers for the eight endpoints PSoC 3 and PSoC 5 devices contain.

Table 17. USB_SIE_EPx_CR0 Registers

Register	PSoC 3 Address	PSoC 5 Address
USB_SIE_EP1_CR0	0x600E	0x4000600E
USB_SIE_EP2_CR0	0x601E	0x4000601E
USB_SIE_EP3_CR0	0x602E	0x4000602E
USB_SIE_EP4_CR0	0x603E	0x4000603E
USB_SIE_EP5_CR0	0x604E	0x4000604E
USB_SIE_EP6_CR0	0x605E	0x4000605E
USB_SIE_EP7_CR0	0x606E	0x4000606E
USB_SIE_EP8_CR0	0x607E	0x4000607E

PSoC Logical Transfer Modes (PSoC 3 Only)

One of the many benefits of PSoC 3 is its ability to support different transfer modes to the USB hardware. In all of the PSoC USB application notes, the CPU has been used to move data into the endpoints to prepare for transfer. However, there is another option: use the DMA to move data into and out of the USB memory. The biggest benefit of this is that the DMA can move the data much faster than the CPU, which increases the capable throughput of the USB device.

Logical transfer modes are a combination of memory management and DMA/CPU configurations. PSoC 3 and PSoC 5 have a 512 byte SRAM that is shared by 8 data endpoints (EP1 to EP8) and 1 control endpoint (EP0). The management of this memory is done either automatically by PSoC or manually by the developer. These modes relate to data transfer within the USB block (data from SRAM to each endpoint) and are unrelated to the various transfer types discussed in this application note (such as, Interrupt, Bulk, Isochronous, and Control).

The PSoC 3 TRM mentions two types of logical transfer modes as shown in Table 18.

Table 18. USB Transfer Modes

Feature	Store and Forward Mode (PSoC 3 and PSoC 5)	Cut Through Mode (PSoC 3 Only)
SRAM Memory Usage	Requires More Memory	Requires Less Memory
SRAM Memory Management	Manual	Automatic
SRAM Memory Sharing	512 bytes of SRAM shared between endpoints. Sharing is done by firmware.	Each endpoint is allocated less share of memory automatically by the block. Rest of memory is available as “Common Area.” This Common Area is used during the transfer.
IN Command	Entire packet present in SRAM memory before the IN command is received.	Memory filled with data only when SRAM IN command is received. Data is given to host when enough data is available (based on DMA configuration). Does not wait for the entire data to be filled.
OUT Command	Entire packet is written to SRAM memory on OUT command. After entire data is available, it is copied from SRAM memory to the USB device.	Waits only for enough bytes (depends on DMA configuration) to be written in SRAM memory. Once enough bytes are present, it is immediately copied from SRAM memory to the USB device.
Transfer of Data	Data is transferred when all bytes have been written to the memory.	Data is transferred when enough bytes are available. It does not wait for the entire data to be filled.
Types Based on DMA	No DMA mode Manual DMA mode	Only Automatic DMA mode
Recommended Transfer	Best suited for Interrupt and Bulk transfers	Best suited for Isochronous transfer

PSoC supports a maximum packet size of 64 bytes using the Store and Forward mode and a maximum packet size of 1023 bytes, which is the limitation of a full speed isochronous transfer, using the Cut Through mode.

As mentioned in Table 15, the Cut Through mode is an automatic memory management mode with automatic DMA access. The CPU performs initial configuration and then transfers control to the USB hardware to control partitioning of the 512 bytes of SRAM and handle the pointers to that memory. Each active IN endpoint is allocated a small amount of memory using the BUF_SIZE register. The remaining memory, after all the IN endpoints have been allocated, is referred to as “common space memory”.

When an IN transfer request is made by the host, the device responds by transferring the data present in the dedicated memory space for that specific endpoint. At the same time, the device issues a DMA request for more data to be moved to that memory space. This data fills up the common space memory. The device does not wait for the entire data payload to be available. For more information regarding USB Logical Transfer Modes, refer to the PSoC 3 TRM.

Project Files

Important: Be sure to not change the VID or PID in the project files. The GUI applications used to test the projects are hard coded with the VIDs and PIDs described. Changing these parameters breaks functionality.

The files included with this application note are WHQL certified to eliminate driver install issues in Windows environments. While INF files are included as references, it is recommended to use the files included with the associated project files to ensure an issue free driver install.

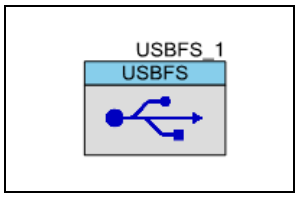
Project 1: Implementing Multiple Transfer Types

This project demonstrates how to implement interrupt, bulk, and isochronous endpoints into a design. The application reads data received in an OUT endpoint for a particular transfer, and perform a loopback by loading that data into a corresponding IN endpoint. This project also demonstrates how to implement Alternate Settings for a single interface in a design.

Configuring the Project

1. Start by opening an empty PSoC Creator project and name it "***Project 1 – USB Transfers***".
2. Locate the USBFS component in the Component Catalog in PSoC Creator and drag it onto the schematic page (***TopDesign.cysh***).

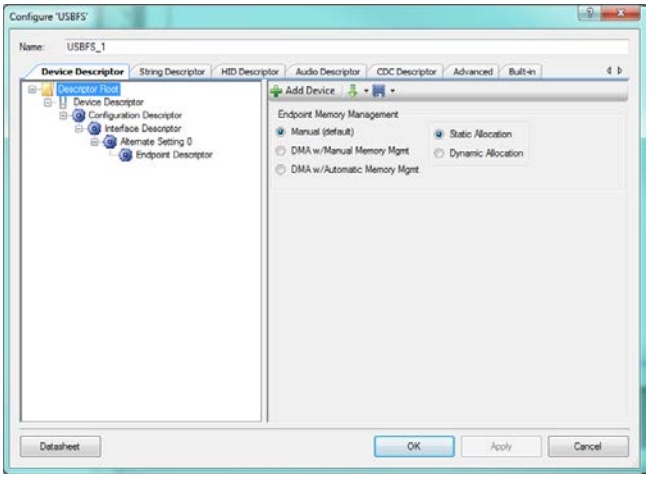
Figure 43. Project 1 – Schematic



3. Right click on the USBFS_1 component and click "**Configure"**. After the configuration window opens, click on "**Descriptor Root**". Ensure that the selections are made as shown in Figure 44:

- Endpoint Memory Management: Manual with Static Allocation

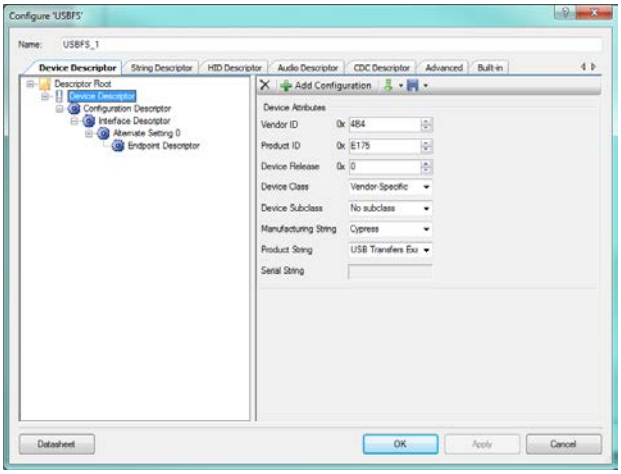
Figure 44. Project 1 – Memory Management Mode Config



4. Next, configure the device descriptors for this USB design. Click on "**Device Descriptor**" and configure the options in the Device Descriptor (see Figure 45). Make the following adjustments:

- Vendor ID: 04B4
- Product ID: E175
- Device Release: 0
- Device Class: Vendor Specific
- Device Sub Class: No Subclass
- Manufacturing String: Cypress
- Product String: USB Transfer Examples

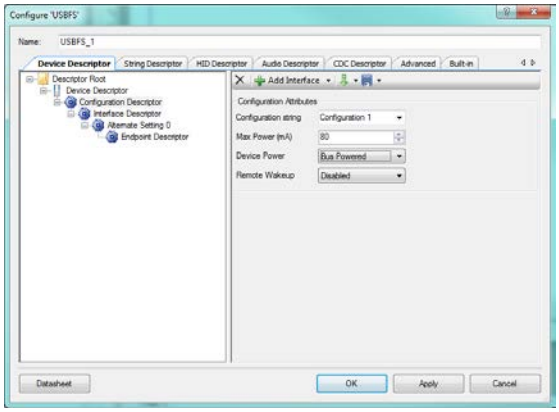
Figure 45. Project 1 – Device Descriptor



5. Next, click on '**Configuration Descriptor'** and adjust the configuration options as follows (see Figure 46):

- Configuration String: Configuration 1
- Max Power: 80
- Device Power: Bus Powered
- Remote Wakeup: Disabled

Figure 46. Project 1 – Configuration Descriptor



In this example, there is a single interface, whose function is to loopback data. To implement different interfaces for the three transfer types, alternate settings will be used. Each setting has a pair or Interrupt, Bulk, and Isochronous IN/OUT endpoints. Therefore, all three transfer type pairs are never simultaneously active.

Using alternate settings for these endpoints does two things:

- It allows you to easily switch between the transfer types you want to use.
- It is easier on the bus bandwidth. Rather than having an interrupt and isochronous endpoints reserving bandwidth they are not using, you can use the alternate settings to only reserve the bandwidth that you use.

For this example, you will reserve bandwidth for 8 bytes on each endpoint. While this is not enough to cause issues with bus bandwidth, this is a good practice to get in the habit of with regard to USB development.

6. Click on the "**Interface Descriptor**" in the USB configuration wizard and then click on "Add Alternate Setting" (see Figure 47) thrice to add three additional alternate settings to the design. The end result should look like what is shown in Figure 48.

Figure 47. Project 1 – Adding Alternate Settings

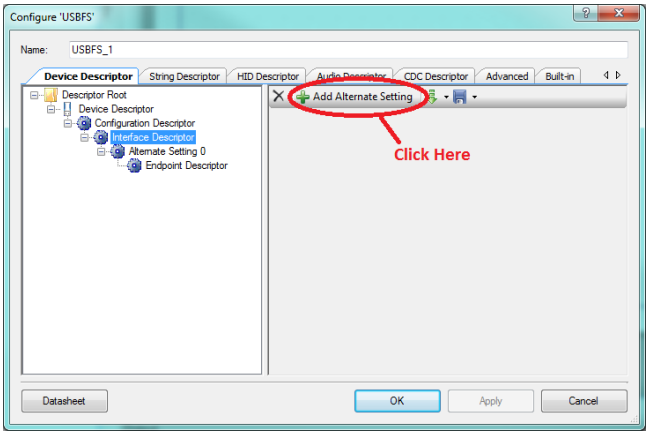
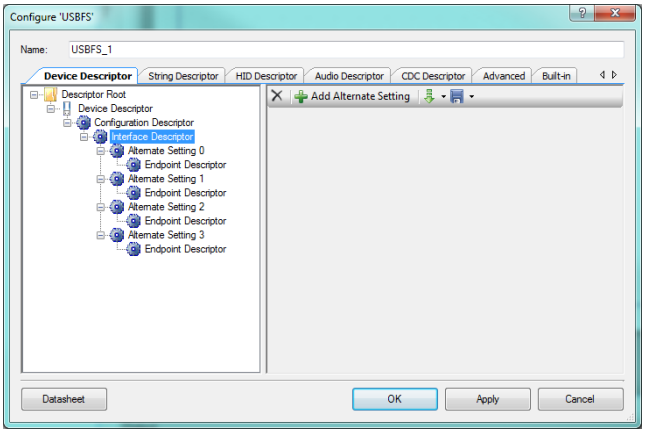
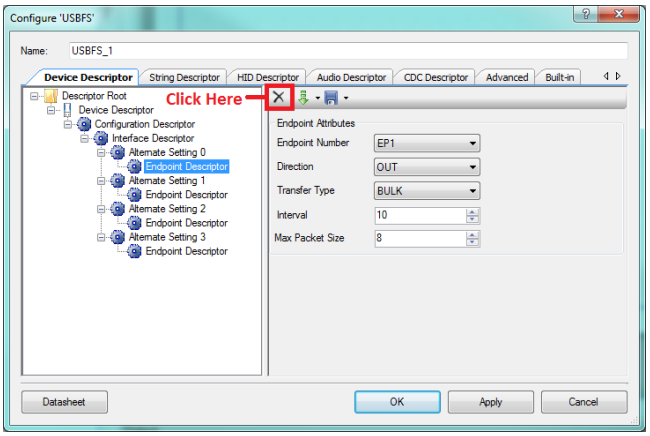


Figure 48. Project 1 – After Adding Alternate Settings



7. Click on the "**Endpoint Descriptor**" located under "Alternate Setting 0" and click on the 'X' to remove the endpoint descriptor (see Figure 49).

Figure 49. Project 1 – Removing EP from Alt. Setting 0

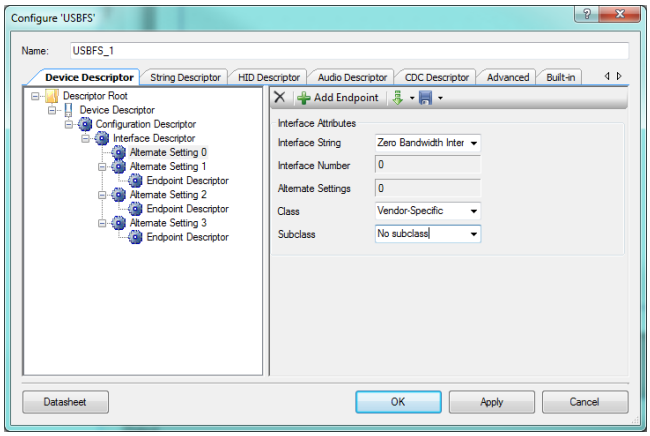


This endpoint must be removed because devices with isochronous endpoints are required to have a zero bandwidth setting to free up bus availability when the isochronous endpoints are not in use.

8. Click on "**Alternate Setting 0**" in the configuration wizard and apply the following configuration options (see Figure 50).

- **Interface String:** Zero Bandwidth Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

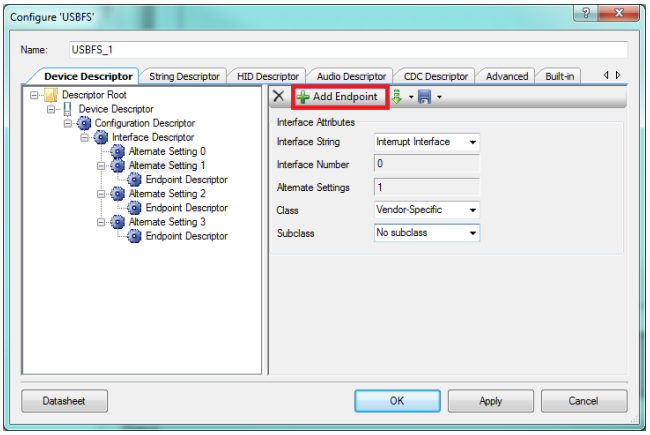
Figure 50. Project 1 – Alternate Setting 0 Configuration



9. Click on "**Alternate Setting 1**" in the configuration wizard and apply the following configuration options (see Figure 51). Additionally, click on the "Add Endpoint" button to add a second endpoint to Alternate Setting 1.

- **Interface String:** Interrupt Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

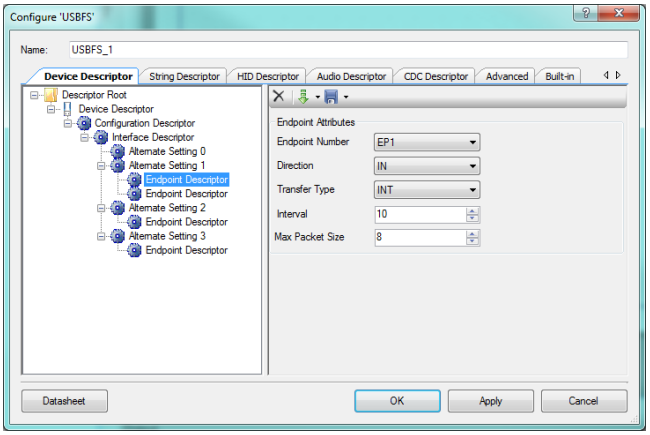
Figure 51. Project 1 – Alternate Setting 1 Configuration



10. Click on the first Endpoint Descriptor under Alternate Setting 1 and configure it based on the following settings (see Figure 52).

- Endpoint Number: EP1
- Direction: IN
- Transfer Type: INT
- Interval: 10
- Max Packet Size: 8

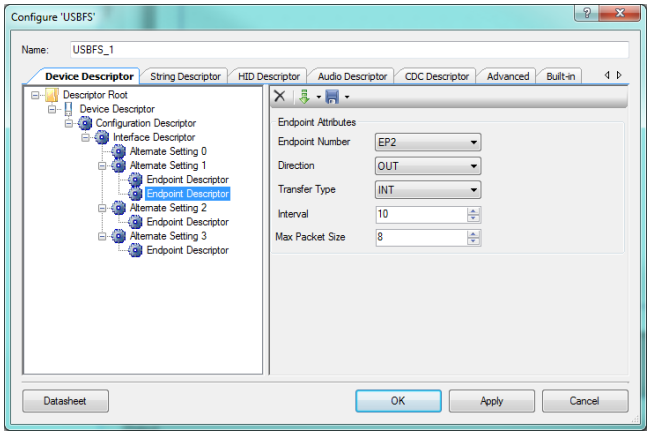
Figure 52. Project 1 – Endpoint 1 Configuration



11. Now click on the second Endpoint Descriptor under Alternate Setting 1 and configure it based on the following settings (see Figure 53).

- Endpoint Number: EP2
- Direction: OUT
- Transfer Type: INT
- Interval: 10
- Max Packet Size: 8

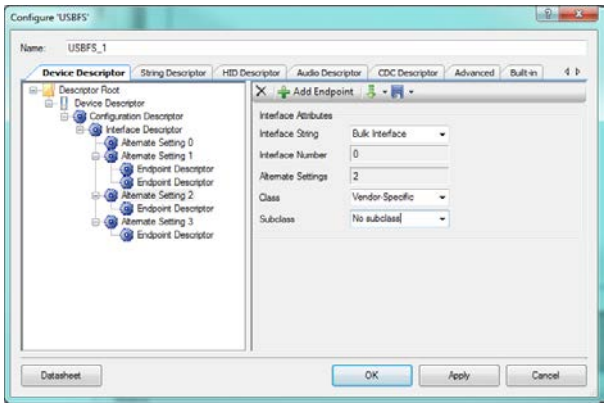
Figure 53. Project 1 – Endpoint 1 Configuration



12. Click on "**Alternate Setting 2**" and add an additional endpoint descriptor (similar to step 9). Apply the following configuration options (see Figure 54) to Alternate Setting 2.

- Interface String: Bulk Interface
- Class: Vendor Specific
- Subclass: No Subclass

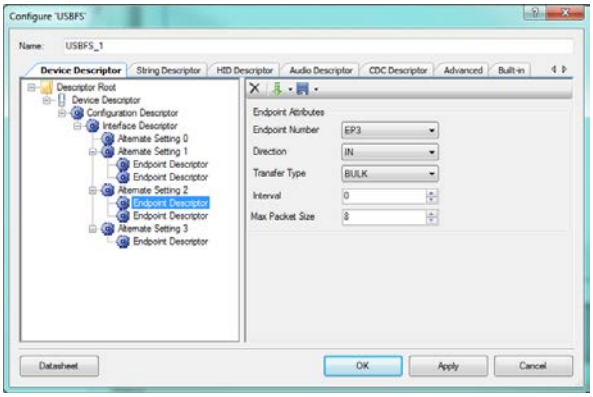
Figure 54. Project 1 – Alternate Setting 2 Configuration



13. Click on the first Endpoint Descriptor under Alternate Setting 2 and configure it based on the following settings (see Figure 55).

- Endpoint Number: EP3
- Direction: IN
- Transfer Type: BULK
- Interval: 0 (NA)
- Max Packet Size: 8

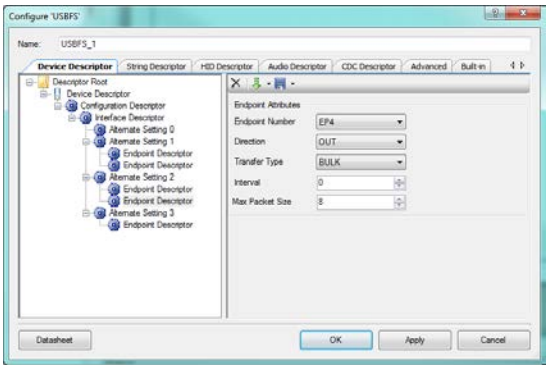
Figure 55. Project 1 – Endpoint 3 Configuration



14. Now click on the second Endpoint Descriptor under Alternate Setting 2 and configure it based on the following settings (see Figure 56).

- Endpoint Number: EP4
- Direction: OUT
- Transfer Type: BULK
- Interval: 0 (NA)
- Max Packet Size: 8

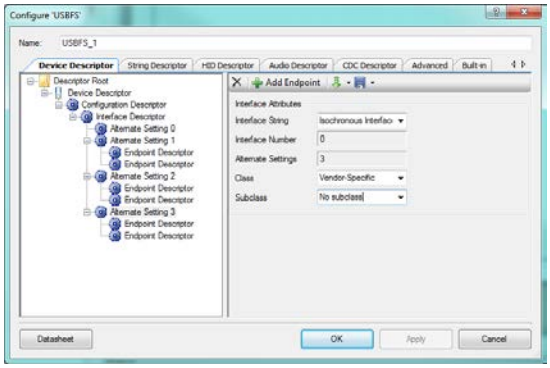
Figure 56. Project 1 – Endpoint 4 Configuration



15. Finally, configure the third alternate setting. Click on "**Alternate Setting 3**" and add an additional endpoint descriptor just as you did in the previous step. Apply the following configuration options (see Figure 57) to Alternate Setting 3.

- **Interface String:** Isochronous Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

Figure 57. Project 1 – Alternate Setting 3 Configuration

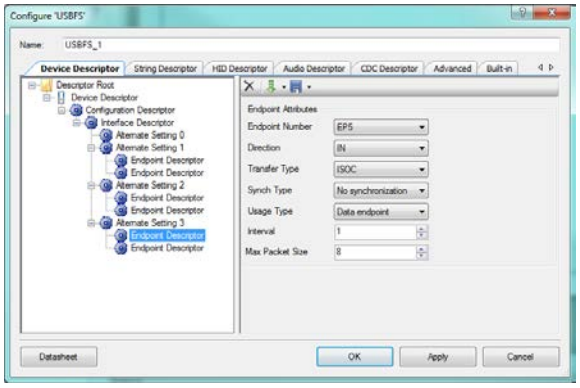


16. Click on the first Endpoint Descriptor under Alternate Setting 3 and configure it based on the following settings (see Figure 58).

- Endpoint Number: EP5
- Direction: IN
- Transfer Type: ISOC
- Synch Type: No Synchronization
- Usage Type: Data Endpoint
- Interval: 1
- Max Packet Size: 8

Since you are only going to stream data to the PC, you do not need to synchronize the PSoC to another device. Therefore, choose "No Synchronization" and configure the endpoint as a data endpoint rather than a feedback endpoint.

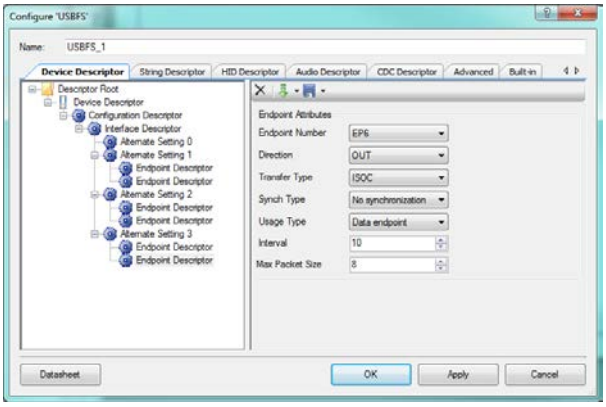
Figure 58. Project 1 – Endpoint 5 Configuration



17. The last thing to configure in the wizard is the OUT ISOC endpoint. Click on the second Endpoint Descriptor under Alternate Setting 3 and configure it based on the following settings (see Figure 59).

- Endpoint Number: EP6
- Direction: OUT
- Transfer Type: ISOC
- Synch Type: No Synchronization
- Usage Type: Data Endpoint
- Interval: 1
- Max Packet Size: 8

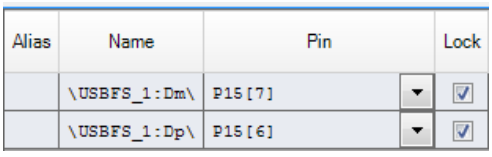
Figure 59. Project 1 – Endpoint 6 Configuration



18. After all these changes are complete, click "**Apply**" and then "**OK**" to close out the configuration wizard.

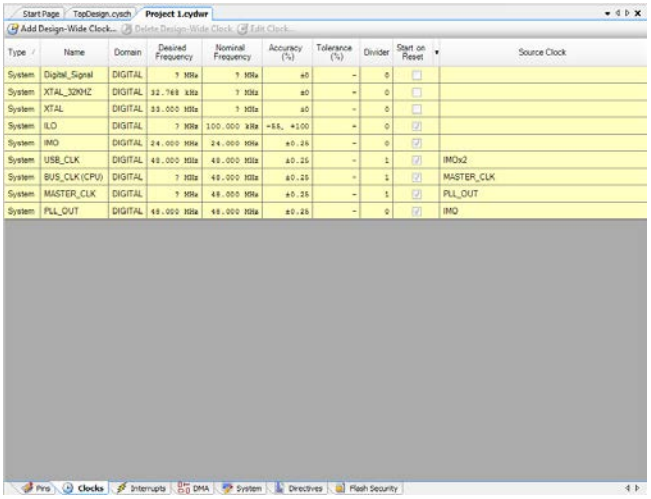
19. Next, you must configure the pins and clocks for the project. In the Workspace Explorer in PSoC Creator, locate "***Project 1 - USB Transfers.cydwr***" and double click on it to open a new configuration tab. The first item that appears is the Pin Configuration. On the upper right hand side of the screen, there are two pins for the D+ and D- lines (see Figure 60). You can manually assign the D+ P15[6] and D- to P15[7]. However, since these are the only two pins that these signal lines can be assigned to, this is a step that can be skipped as PSoC Creator automatically assigns the pins to the proper location upon building the project.

Figure 60. Project 1 – Pin Configuration



20. Click on the "**Clocks**" tab at the bottom of the screen. You will see a screen showing various rows and columns of clocks as shown in Figure 61. Double click on any of the clocks to open the clocks configuration wizard.

Figure 61. PSoC Creator Clocks

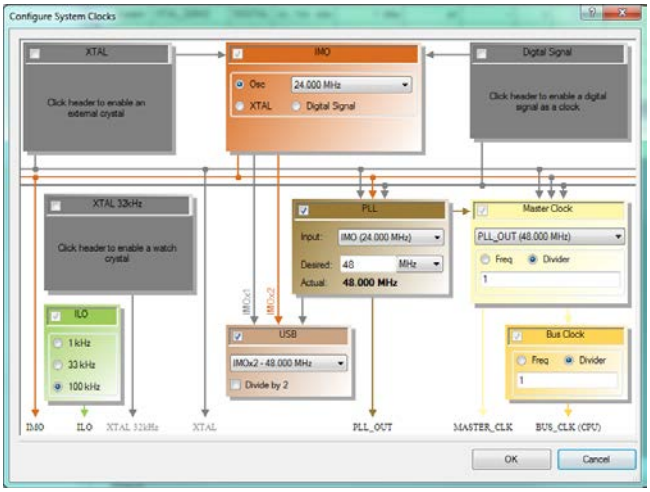


21. Once you open the configuration wizard, adjust the clocks to the following specifications (see Figure 62):

- **IMO**: 24 MHz
- **ILO**: 100 kHz
- **PLL**: 48 MHz
- **USB**: Check box to enable, set to IMOx2 - 48MHz
- **Master** Clock: PLL Out

Note On PSoC 5 devices, you will be required to use a 24 MHz ECO for USB operation. In that instance, ensure that the "XTAL" box is checked and configured for 24 MHz. In addition, ensure the IMO is set to source from "XTAL" rather than "Osc".

Figure 62. Clocks Configuration Wizard



Click "**OK**" upon completion of this step.

22. The last step that needs to be performed prior to programming out design into a PSoC is to place some code into *main.c*. Located in **Appendix B** is the code for Project 1. Double click on the *main.c* file, located under source files in the Workspace Explorer. Place this code in *main.c*. Comments are located in **Appendix B** also to aid in understanding the flow of the code.

The GUI application used with this project is the Cypress USB Control Center. When you want to activate the different Alternate Settings, the Control Center issues the USB request to switch over to that specific setting, which we will look at later. The firmware in the PSoC device checks for the change in interface setting (alternate setting) and executes the proper code for that alternate setting by polling, reading, and loading the proper endpoints. Additionally, when information is received on the OUT endpoint, the firmware loads that data into the paired IN endpoint to be read by the PC.

23. The next step is to build the project so that it can be programmed into the device. In PSoC Creator, locate the Project drop down menu at the top of the screen. Select **Build > Build Project 1 – USB Transfers**. After the project is built without errors or warnings, program a PSoC 3 or PSoC 5 device.

Testing the Project

One of the prerequisites of this application note is to download Cypress Suite USB v3.4.6 or later. Navigate to the application folder from your Windows Start menu and open Control Center. You can find the application under **All Programs > Cypress > Cypress Suite USB 3.4.6 > Control Center.**

24. Plug the PSoC into an open USB port on a PC. Since PSoC is configured as a Vendor Specific device, Windows asks for a driver (*.inf* and *.sys*). Navigate windows to the "**Driver**" folder, included with the project files for this application note.

If you do not have access to the driver files, locate the INF file in **Appendix A**. Open an empty Notepad document in Windows and place the text in **Appendix A** in the document. Save the file as *AN56377.inf*.

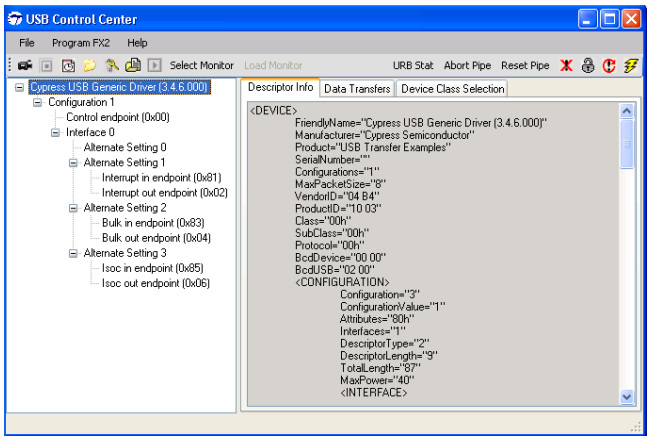
When Windows asks for the *.sys* file, navigate to the Cypress Suite USB folder. Depending on your operating system, the path should be similar to the following:

```
C:\Cypress\Cypress Suite USB 3.4.6\Driver\bin
```

From that point, navigate further to the appropriate operating system (such as Windows XP or Windows 7) and Operating System type (32-bit or 64-bit). Remember that when following this second method, the driver will not be signed and may prevent the driver from being installed properly. It is recommended to use the drivers that were downloaded with the associated project files.

25. After the PSoC has been enumerated, open USB Control Center, and expand out all the tabs on the left hand side of the application. See Figure 63.

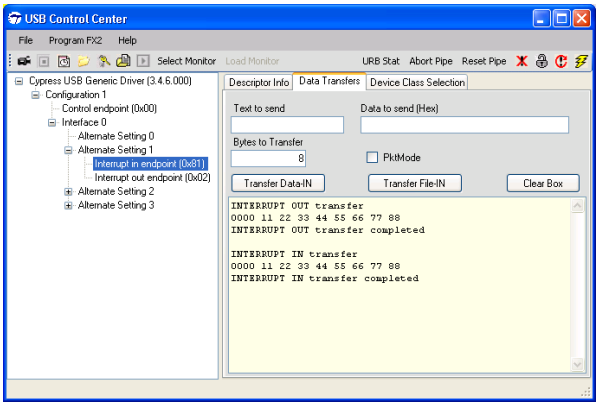
Figure 63. Project 1 in USB Control Center



Note that there is a zero bandwidth alternate setting (Alternate Setting 0) and three other alternate settings (Alternate Setting 1 through Alternate Setting 3) with the various endpoint types. Upon enumeration, Alternate Setting 0 is selected by default.

26. Select "**Interrupt out endpoint (0x02**)", which is located under "Alternate Setting 1", and click on the "**Data Transfers**" tab. In the "**Data to send (Hex)**" text field, go ahead and place 8 hex bytes, followed by clicking "**Transfer Data-OUT**". Then click on "**Interrupt in endpoint (0x81)**" and click the "**Transfer Data-IN**" button. Note that a successful IN and OUT transfer occurs in the status window (see Figure 64).

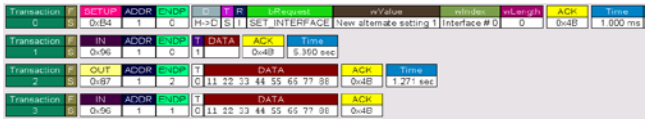
Figure 64. Testing Project



Going a step further and looking at a Bus Analyzer trace, note that a control transfer also took place prior to the loopback data transfer. The control transfer switched the alternate setting from Alternate Setting 0 to Alternate Setting 1. Figure 65 shows the following information:

- **Transaction 0**: Setup Stage
- **Transaction 1**: Status Stage (No data stage since wLength was set to 0).
- **Transaction 2**: OUT Transaction (Host to Device)
- **Transaction 3**: IN Transaction (Device to Host)

Figure 65. Project 1 on a Bus Analyzer



27. You can experiment with the other transfer types. The general concept is similar in terms of implementation. Isochronous can be more complex, however, depending on the configuration. While using the USB Control center, you may transfer less than eight bytes when using interrupt and bulk. With the isochronous transaction, you will need to transfer exactly eight bytes. This is not a limitation of USB but intended functionality of the CYUSB driver. Be sure to fill in the "Bytes to Transfer" fields appropriately.

Figure 66 shows the three various transfer types (Interrupt, Bulk, and Isochronous) and an OUT transfer for each one, performed using this code example. You can see the three control transfers that change the Alternate Setting. Also note the lack of a handshake packet in the isochronous transaction, shown in Transaction 8.

Figure 66. Various USB OUT Transactions



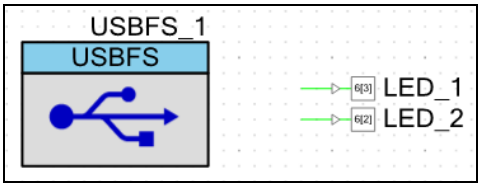
Project 2: Implementing Vendor Commands

This project demonstrates how to implement vendor commands into a USB design and how to transfer data using a control transfer. This code example uses vendor commands to turn LEDs on and off using a PSoC DVK board.

Configuring the Project

1. Start by creating a new PSoC Creator project and name it **Project 2 – USB Control Transfers**.
2. Locate the USBFS component in the Component Catalog in PSoC Creator and drag it to the schematic page (*TopDesign.cysh*) along with two digital output pins. (In this example, they are named LED_1 and LED_2). You can name them anything, but make certain to modify the code that will be provided later.

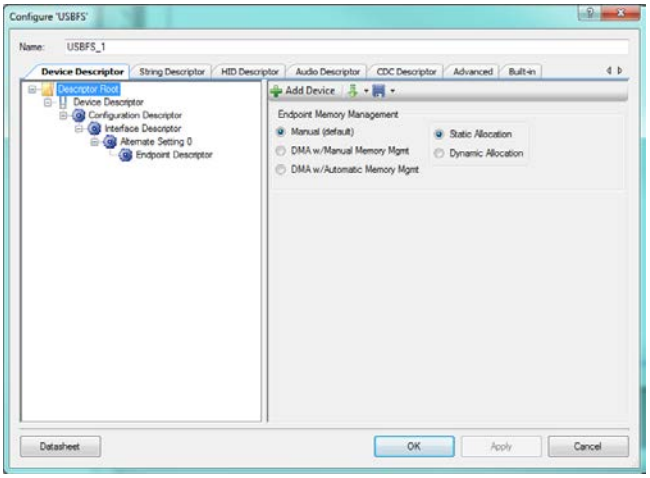
Figure 67. Project 2 - Schematic



3. Right click on the USBFS component and click "**Configure**". After the configuration window opens, click on "**Descriptor Root**". Ensure that the following selections are made (see Figure 68).

- Endpoint Memory Management: Manual with Static Allocation.

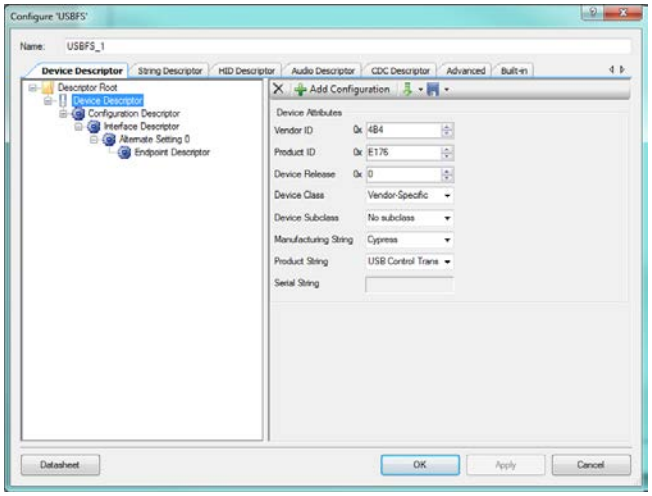
Figure 68. Project 2 – Memory Management Mode Config



4. Next, configure the device descriptors for this USB design. Click on "**Device Descriptor**" and configure the component (see Figure 69). The settings to be made are as follows:

- Vendor ID: 04B4
- Product ID: E176
- Device Release: 0
- Device Class: Vendor Specific
- Device Sub Class: No Subclass
- Manufacturing String: Cypress
- Product String: USB Control Transfer Example

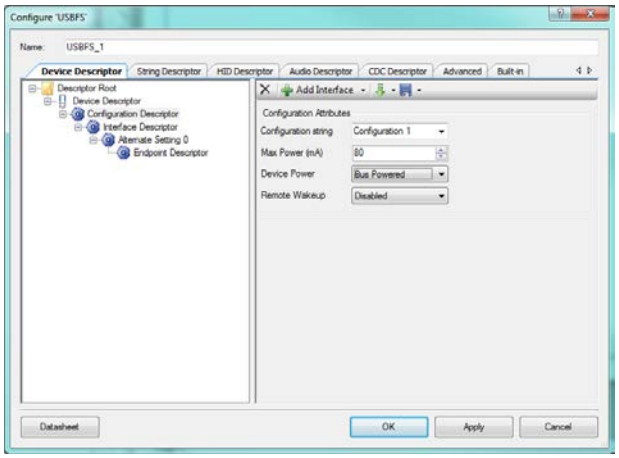
Figure 69. Project 2 – Device Descriptor



5. Next, click on the '**Configuration Descriptor** and adjust the configuration options as follows (see Figure 70).

- Configuration String: Configuration 1
- Max Power: 80
- Device Power: Bus Powered
- Remote Wakeup: Disabled

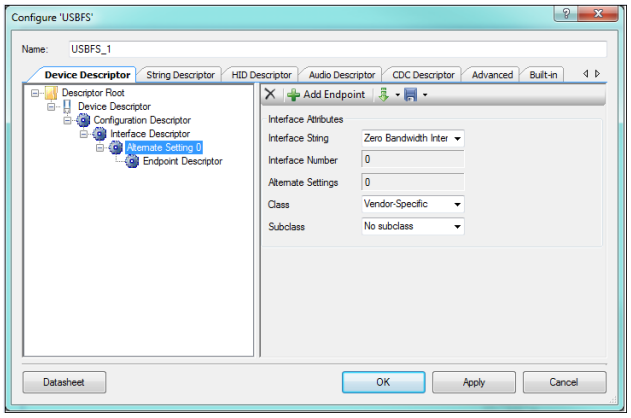
Figure 70. Project 2 – Configuration Descriptor



6. Click on "**Alternate Setting 0**" and configure it to the following parameters (see Figure 71).

- **Interface String:** Zero Bandwidth Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

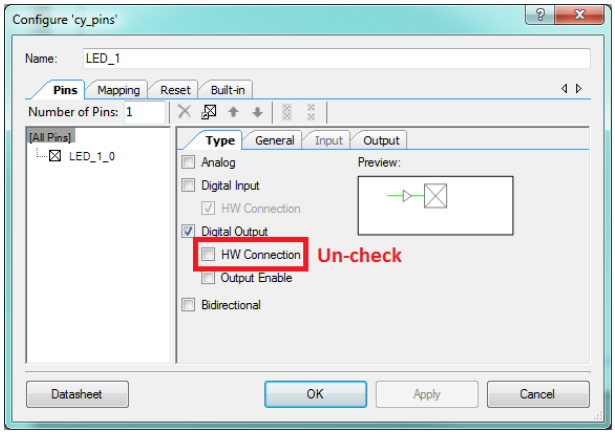
Figure 71. Project 2 – Alternate Setting 0 Configuration



7. After all these changes are complete, click "**Apply**" and then "**OK**" to close the configuration wizard. Since you do not use an IN or OUT endpoint, there is no need to worry about configuring the Endpoint Descriptor.

8. Next, double click on one of the digital output pins to open the configuration wizard. Leave the pins in their default settings with the exception of unchecking the "**HW Connection**" box located in the "**Type**" tab on the configuration wizard (see Figure 72). Apply the same change to the other digital output pin.

Figure 72. Project 2 – Digital Output Pin Configuration



9. Just as was done in Project 1, configure the pins and clock settings for this project. Double click on **Project 2 - USB Control Transfers.cydwr** to open the configuration menu.

10. Beginning with the pin configuration, place the two LED pins at GPIO locations to which you can easily attach two LEDs. If you do not have LEDs , you can use a DMM or an oscilloscope to observe the results. In this example, a CY8CKIT-030 was used with

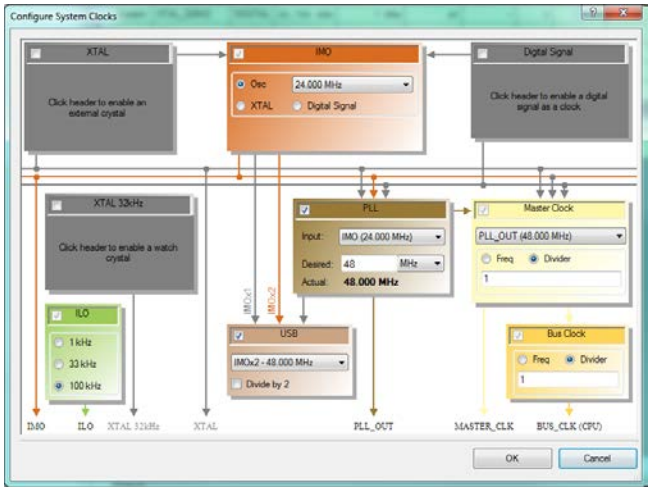
LED_1 placed on P6[3] and LED_2 on P6[2]. Modify these pin locations if you are using a CY8CKIT-001 or any other Cypress evaluation kit.

Figure 73. Project 2 – Pin Placement

Alias	Name	Pin	Lock
	\USBFS_1:Dm\	P15[7]	☑
	\USBFS_1:Dp\	P15[6]	☑
	LED_1	P6[3]	☑
	LED_2	P6[2]	☑

11. Next, click on the "**Clocks**" tab at the bottom of the .cydwr menu and configure the clocks in the same manner as in steps 20 and 21 in Project 1 (see Figure 74). Remember that the same requirement for a 24 MHz ECO applies if you use PSoC 5.

Figure 74. Clocks Configuration Wizard



12. Now that the components are configured, place the required C code in the design. There are two files that require editing. Start by opening *main.c*, which can be found in the workspace explorer. Place the C code found in **Appendix D** of this application note into *main.c*.

13. Some of the files that required editing do not yet exist until you build the project for the first time. Select **Project > Build Project 2 – USB Control Transfers** from the drop down menu to build the project. After the project is successfully built, look for the USBFS_1 folder in the PSoC Creator workspace explorer. Inside that folder is a file named *USBFS_1_vnd.c*. This file stores the code to handle vendor commands, as 'vnd' is an abbreviation for vendor. Double click on *USBFS_1_vnd.c* to open it.

14. There are two locations in *USBFS_1_vnd.c* where code must be placed. At approximately line number 26, you should see the following text (see Figure 75).

Figure 75. USBFS_1_vnd.c Custom Declarations

```
21
22  /***************************************
23   * Vendor Specific Declarations
24   ***************************************/
25
26  /* `#START VENDOR_SPECIFIC_DECLARATIONS` Place your declaration here */
27
28  /* `#END` */
29
```

In between the "#START" and "#END" place the code found in the first half of **Appendix C**, located in Code 1. Note that that if you changed the overall name of the USBFS component, you must edit the name of the **USBFS_1_InitNoDataControlTransfer** function prototype seen in **Appendix C**.

15. Scroll down further in the *USBFS_1_vnd.c* file to approximately line number 95. Note another section to place the customer code in the file (see Figure 76).

Figure 76. USBFS_1_vnd.c Custom Code

```
93      }
94
95      /* `#START VENDOR_SPECIFIC_CODE` Place your vendor specific request here */
96
97      /* `#END` */
```

In between the "#START" and "#END" place the code found in the second half of **Appendix C**. The code in **Appendix C** implements the functionality to do something with the control data that is received by the device. When a control transfer is performed, variables such as bmRequestType, wIndex, and others, are stored in a register location. This code will read out the relevant information from those registers and perform the appropriate function based on the information received. See the comments in the **Appendix C** code for additional details.

There are four vendor commands implemented in this application. These values are the bRequest discussed in the control transfer section earlier in the application note. The four commands, which are implemented, are the following:

- SET_LED = 0xA1
- CLEAR_LED = 0xB1
- CONTROL_READ = 0xA2
- CONTROL_WRITE = 0xB2

When using the SET_LED and CLEAR_SET to turn the LED on and off, wValue will be used to indicate which LED you wish to control by using either 0x0001 for LED_1 and

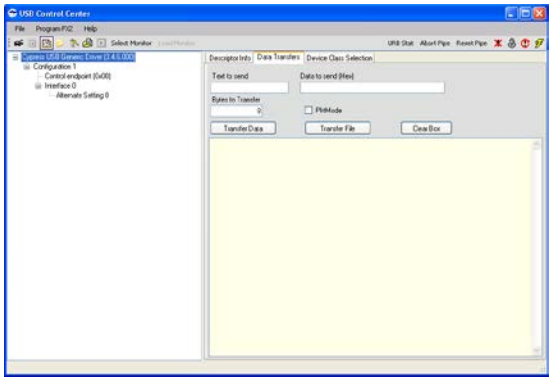
0x0002 for LED_2. The variable for wIndex is not used in this application. SET_LED and CLEAR_LED will demonstrate how to implement a no data phase control transfer. CONTROL_READ and CONTROL_WRITE will demonstrate how to send and receive data over a control transfer by performing a control transfer with a data phase.

16. Now we are ready to test this project. Go ahead and rebuild the project. After the project is built without errors or warnings, program a PSoC 3 or PSoC 5 device.

Testing the Project

17. Plug the PSoC development board into the PC through a USB cable. You will need to go through the same enumeration procedure with installing the driver, which was done in Project 1. Once the device is plugged into the PC, powered, and fully enumerated, open the Control Center application just as was done in Step 25 of Project 1. Expand the configuration tree for the device by clicking the "+" on left side of the application. The application window should look similar to Figure 77.

Figure 77. Project 2 – Control Center View



18. Click on "**Control endpoint (0x00)"** in Control Center, and then click on the "**Data Transfers**" tab. This is where you will issue requests to the device. Start by making the following adjustments to the GUI.

- Bytes to Transfer: 0
- Direction: OUT
- Req Type: Vendor
- Target: Device
- Request Code: 0xA1
- wValue: 0x0001
- wIndex: 0x0000

19. Click on the "**Transfer Data**" button and you should see the Control Center application confirm a zero-length data transfer was completed successfully as shown in Figure 81. Additionally, note that the LED on your DVK board is illuminated. If you change the value in Req Code to 0x00B1, then press "**Transfer Data"** again, and the LED will turn off again. You can also experiment with turning the other LED on and off by changing the wValue from 0x0001 to 0x0002. If we look at a bus analyzer trace, shown in Figure 78, we can see the presence of a setup stage and a status stage. Note there is no data stage.

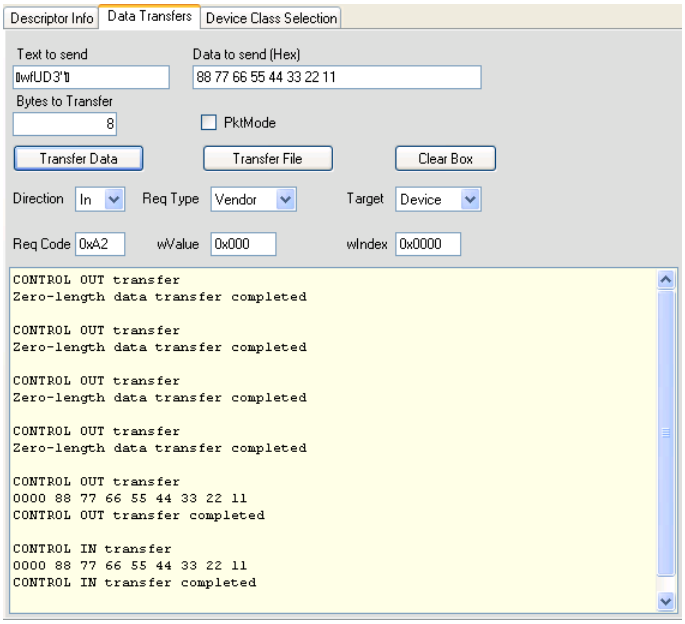
Figure 78. Bus Analyzer Capture for No Data Phase Transfer



20. Next, let us take a look at performing a data control transfer. Start by changing the Req Code to 0xB2, the wValue to 0x0000, and the Direction to Out. In the "Data to Send (Hex)" field, type in eight bytes to send. Then press the "Transfer Data" button. You should see a message saying that the data was successfully transferred.

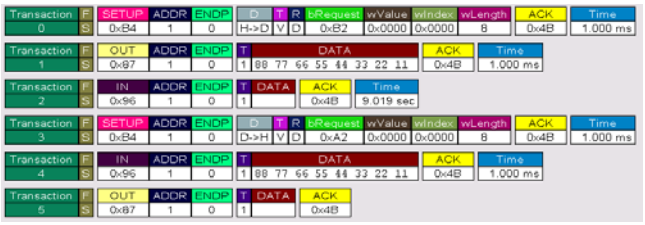
21. Now read the data back from the host. Change the "Direction" back to "In" and change the Req Code to 0xA2. Press the "Transfer Data" button again and the message window displays an echo of the data that was previously sent. Figure 79 shows an example of the expected results.

Figure 79. Project 2 Testing using USB Control Center



If we were to look at data transfer on a bus analyzer, we see a total of six stages: three stages for the write and three stages for the read. This bus analyzer trace can be seen in Figure 81 Notice that transaction 0 though transaction 2 show the setup, data, and status stage for the write, while transactions 3 through 5 show the setup, data, and status stage for the read.

Figure 80. Bus Analyzer Capture for Control Data Transfer



Project 3: Increasing USB Throughput with DMA on PSoC 3

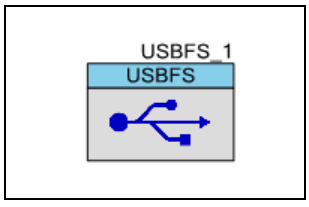
This project demonstrates how to implement the DMA in a USB application to achieve greater throughput and how to implement 1023 byte isochronous transfers for streaming data to the host.

Note This project uses a feature that is only supported on PSoC 3 devices. PSoC 5 does not support this feature.

Configuring the Project

1. Start by creating a new PSoC Creator project and name it **Project 3 – USB DMA Example** and place a USBFS component onto the schematic. Then, open the configuration wizard for USBFS.

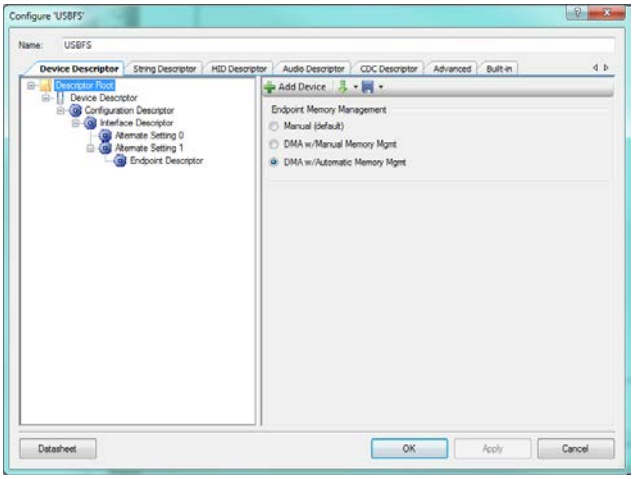
Figure 81. Project 1 - Schematic



2. Right click on the USBFS component and click "**Configure**". Once the configuration window opens, start by clicking on "**Descriptor Root**". Ensure that the following selections are made as seen in Figure 83:

- Endpoint Memory Management: DMA with Automatic Memory Mgmt

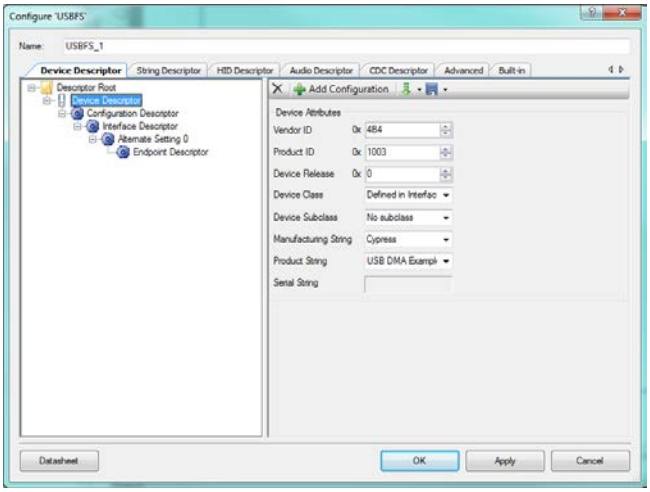
Figure 82. Project 3 – Memory Management Mode Config



3. Configure the device descriptors for this USB design. Click on "**Device Descriptor**" and configure the component as you see in Figure 83. Make the following settings:

- Vendor ID: 04B4
- Product ID: 1003
- Device Release: 0
- Device Class: Vendor Specific
- Device Sub Class: No Subclass
- Manufacturing String: Cypress
- Product String: USB DMA Example

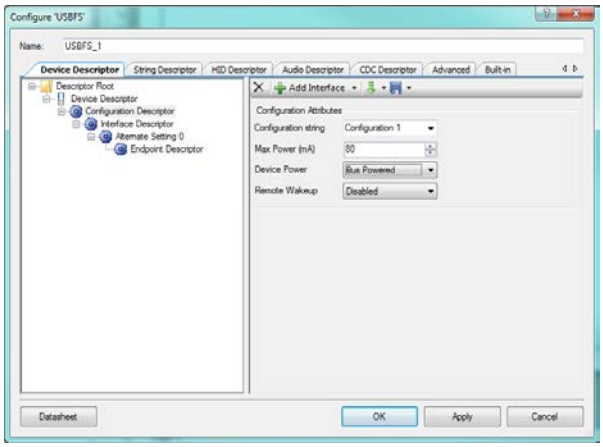
Figure 83. Project 3 – Device Descriptor



4. Click on the '**Configuration Descriptor** and adjust the configuration options as listed below and in Figure 88.

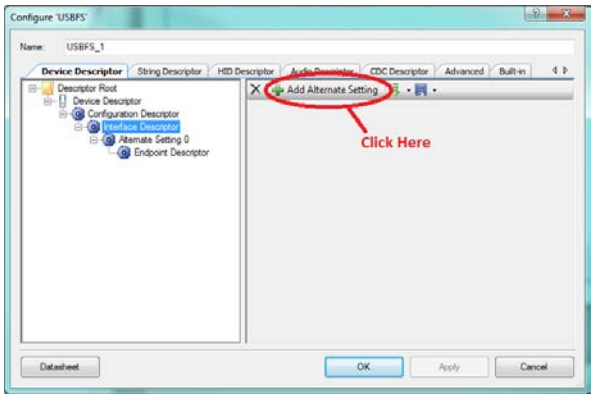
- Configuration String: Configuration 1
- Max Power: 80
- Device Power: Bus Powered
- Remote Wakeup: Disabled

Figure 84. Project 3 – Configuration Descriptor



5. Since an isochronous endpoint that is reserving a bandwidth for 1023 bytes every frame is being implemented, it is important to configure a zero bandwidth alternate setting similar to what was done in Project 1. Click on the **Interface Descriptor** in the USBFS wizard and add an additional alternate setting as shown in Figure 89.

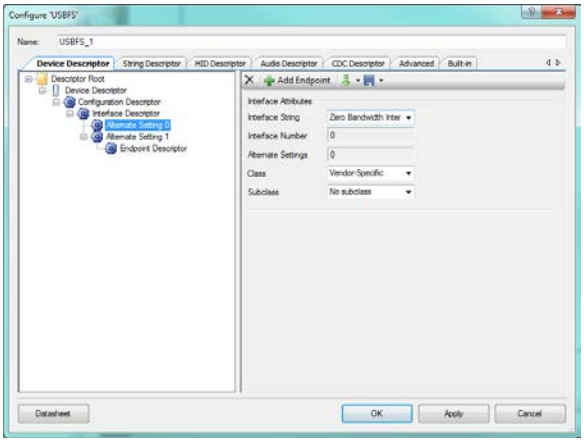
Figure 85. Project 3 – Adding Alternate Settings



6. Remove the endpoint contained in Alternate Setting 0 so that only Alternate Setting 1 contains a single endpoint. Click on "**Alternate Setting 0**" and configure it according to the following settings and as shown in Figure 87.

- **Interface String:** Zero Bandwidth Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

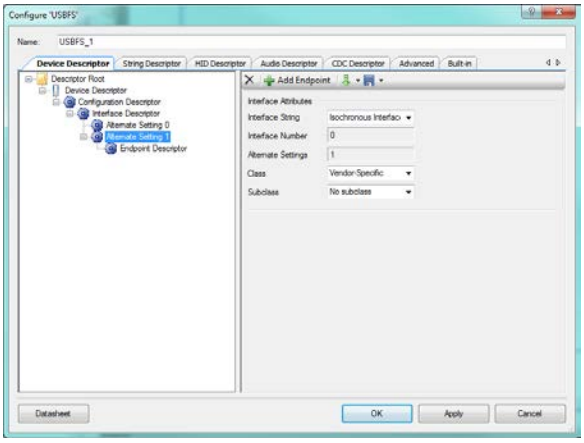
Figure 86. Project 3 – Configuring Zero Bandwidth Interface



7. Next, click on "**Alternate Setting 1**" and configure it according to the following settings and as shown in Figure 88.

- **Interface String:** Isochronous Interface
- **Class:** Vendor Specific
- **Subclass:** No Subclass

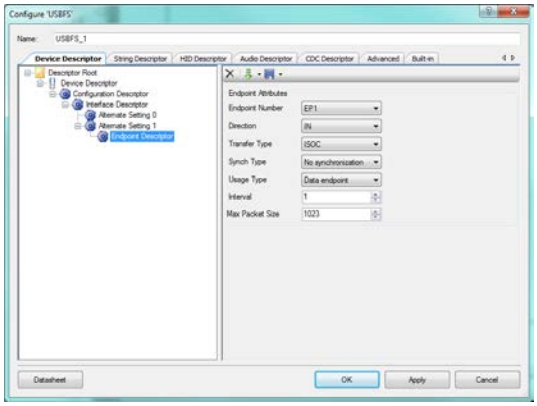
Figure 87. Project 3 – Configuring Alternative Setting 1



8. The final step in configuring the USBFS component is to configure the endpoint. This code example consists of a single isochronous IN endpoint. Click on the "Endpoint Descriptor" and configure it based on the following parameters and as shown in Figure 88.

- Endpoint Number: EP1
- Direction: IN
- Transfer Type: ISOC
- Synch Type: No Synchronization
- Usage Type: Data Endpoint
- Interval: 1
- Max Packet Size: 1023

Figure 88. Project 3 – Endpoint Descriptor



Keep in mind that achieving 1023 byte isochronous transfers in PSoC 3 is only achievable by using automatic DMA mode since the endpoint RAM is limited to 512 bytes.

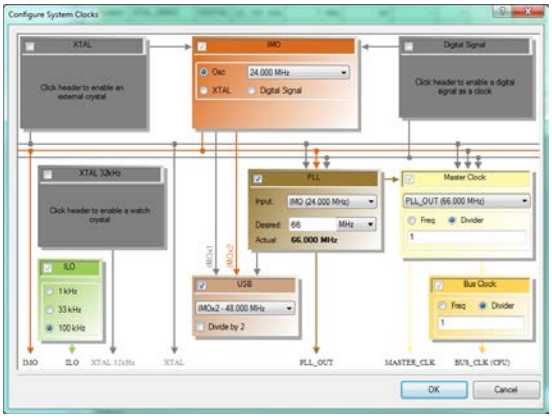
9. Once these changes have been made, click "**Apply**" and "**OK**" to save and close the configuration wizard.

10. Open the **Project 3 – USB DMA Example.cydwr** file. The pins can be left alone since we only have the D+ and D- pins in this design. They will be auto assigned by the router upon building the project.

11. Proceed to configure the clocks the same way that was done in Project 1 and Project 2. However, make one small change to the PLL as shown in Figure 90.

- **PLL:** 66 MHz

Figure 89. Project 3 – Clocks Wizard



12. Apply these changes and proceed to the *main.c* file. Locate the code for Project 3 in **Appendix E** and place it into main.c. The code initially configures the DMA using the USBFS_1_LoadInEP API. From that point on, the DMA takes care of loading data from RAM into the EP buffer as space becomes available based on what was discussed in the PSoC Logical Transfer Modes section. The only other required section of code is to reinitialize the DMA with a null pointer upon completion of a DMA transaction. The remaining code in the application exists to reinitialize the USB in the instance of a configuration change (alternate setting change) or stop execution if the USB becomes unconfigured.

13. Open the USBFS_1_episr.c file and under the ISR for Endpoint 1 (starting at approximately line 60), comment out the following line of code:

```
USBFS_EP[USBFS_EP1].epToggle ^= USBFS_EPX_CNT_DATA_TOGGLE;
```

14. Now we are ready to test this project. Go ahead and rebuild the project. Once the project has been built without errors or warnings, program a PSoC 3 device.

Testing the Project

15. Plug the PSoC into a USB port on a PC. Using the INF provided with the project files (or in **Appendix A**), install the CYUSB driver for this application. For this code example, we will use a different SuiteUSB application. Instead of using the CY Control Center, this project uses the USB Data Streamer application. This application is used to send a constant stream of data to test the throughput. You should find the application under **All Programs > Cypress > Cypress Suite USB 3.4.6 > Streamer.**

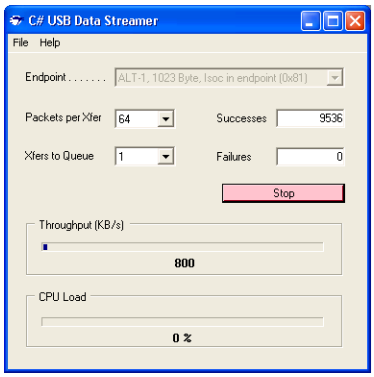
16. Configure the application with the following parameters via the drop down boxes.

- **Endpoint**: ALT-1, 1023 Byte, Isoc in endpoint (0x81)

- **Packets per Xfer**: 64
- **Xfers to Queue**: 1

17. Once all the settings have been made, press the **"Start"** button. You will see an incremental value of successful transfers appearing in the "Success" box shown in Figure 90.

Figure 90. USB Streamer Application



The Streamer application creates a constant data stream to the specific endpoint selected based on the parameters that were chosen. Notice the 800 KB/s transfer occurring. To provide further perspective, the following is a list of test results of various transfer speeds obtained using the different USB Memory Management configurations.

- **Manual (64-byte)**: ~200 KB/s
- **DMA w/ MMM (64-byte):** ~500KB/s
- **DMA w/ AMM (1023-byte)**: ~800KB/s

Additionally, if we look at a bus analyzer capture of the transfers being performed by the Streamer application in Figure 91, we can see that with each SOF that occurs, we perform a 1023 byte data transfer.

Figure 91. Bus Analyzer Trace of Project 3

Transaction	F	IN	ADDR	ENDP	T	DATA
0	S	0x96	1	1	0	1023 bytes

Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
2	S	00000001	0xA5	1475	0x0C	3.00	4

Transaction	F	IN	ADDR	ENDP	T	DATA
1	S	0x96	1	1	0	1023 bytes

Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
5	S	00000001	0xA5	1476	0x12	2.75	4

Transaction	F	IN	ADDR	ENDP	T	DATA
2	S	0x96	1	1	0	1023 bytes

Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
8	S	00000001	0xA5	1477	0x0D	2.75	4

Transaction	F	IN	ADDR	ENDP	T	DATA
3	S	0x96	1	1	0	1023 bytes

Packet #	F	Sync	SOF	Frame #	CRC5	EOP	Idle
11	S	00000001	0xA5	1478	0x0F	3.00	4

Summary

Based on the material and examples in this application note, you should now have an understanding of the basics involved with implementing different transfer types in a USB project, implementing vendor commands into a USB design, and utilizing the DMA for faster data throughput to eliminate the bottle neck of the CPU requiring handling the information. If you want to experiment with the manual DMA mode for USB transfers, or learn additional details about the automatic mode, refer to the USBFS component data sheet in PSoC Creator. Additionally, PSoC Creator includes an example project for creating an Audio Class device which shows how to incorporate synchronous synchronization with isochronous transfers into a USB application.

Related Resources

Application Notes

- **AN57294** – USB 101: An Introduction to Universal Serial Bus
- **AN57473** - PSoC® 3 / PSoC 5 USB HID Fundamentals (with Mouse and Joystick)
- **AN58726** - PSoC® 3 / PSoC 5 USB HID Intermediate (with Keyboard and Composite Device)
- **AN023** - USB Compliance Testing Overview
- **AN52970** - Windows Hardware Quality Lab (WHQL) Signing Procedure
- **AN14557** – Introduction to CYUSB.dll Based Application Development Using C#
- **AN61744** – Introduction to CYAPI.lib Based Application Development Using VC++

Additional Information

- Cypress PSoC USB Landing Page
- Official USB 2.0 Specification
- USB Complete by Jan Axelson

About the Author

Name:	Robert Murphy
Title:	Application Engineer Sr.
Background:	Robert Murphy graduated from Purdue University with a Bachelor's Degree in Electrical Engineering Technology
Contact:	rlrm@cypress.com

Appendix A: INF for AN56377

```
[Version]
Signature="$WINDOWS NT$"
Class=USB
ClassGUID={36FC9E60-C465-11CF-8056-444553540000}
provider=%CYUSB_Provider%
CatalogFile=CYUSB.cat
DriverVer=06/17/2011,3.4.6.000

[SourceDisksNames]
1=%CYUSB_Install%,,,

[SourceDisksFiles]
CYUSB.sys = 1

[DestinationDirs]
CYUSB.Files.Ext = 10,System32\Drivers

[ControlFlags]
ExcludeFromSelect = *

[Manufacturer]
%CYUSB_Provider%=Device,NT,NTx86,NTamd64

;for all platforms
[Device]
%VID_04B4&PID_E175.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E175
%VID_04B4&PID_E176.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E176
%VID_04B4&PID_1003.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_1003


;for windows 2000 non intel platforms
[Device.NT]
%VID_04B4&PID_E175.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E175
%VID_04B4&PID_E176.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E176
%VID_04B4&PID_1003.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_1003


;for x86 platforms
[Device.NTx86]
%VID_04B4&PID_E175.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E175
%VID_04B4&PID_E176.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E176
%VID_04B4&PID_1003.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_1003


;for x64 platforms
[Device.NTamd64]
%VID_04B4&PID_E175.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E175
%VID_04B4&PID_E176.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_E176
%VID_04B4&PID_1003.DeviceDesc%=CyUsb,
USB\VID_04B4&PID_1003


[CYUSB]
CopyFiles=CYUSB.Files.Ext
AddReg=CyUsb.AddReg

[CYUSB.HW]
AddReg=CYUSB.AddReg.Guid

[CYUSB.Services]
Addservice = CYUSB,2,CYUSB.AddService

[CYUSB.NT]
CopyFiles=CYUSB.Files.Ext
AddReg=CyUsb.AddReg

[CYUSB.NT.HW]
AddReg=CYUSB.AddReg.Guid

[CYUSB.NT.Services]
Addservice = CYUSB,2,CYUSB.AddService


[CYUSB.NTx86]
CopyFiles=CYUSB.Files.Ext
AddReg=CyUsb.AddReg

[CYUSB.NTx86.HW]
AddReg=CYUSB.AddReg.Guid

[CYUSB.NTx86.Services]
Addservice = CYUSB,2,CYUSB.AddService

[CYUSB.NTamd64]
CopyFiles=CYUSB.Files.Ext
AddReg=CyUsb.AddReg

[CYUSB.NTamd64.HW]
AddReg=CYUSB.AddReg.Guid

[CYUSB.NTamd64.Services]
Addservice = CYUSB,2,CYUSB.AddService


[CYUSB.AddReg]
; Deprecating - do not use in new apps to identify a
CYUSB driver
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,CYUSB.sys
; You may optionally include a check for DriverBase in
your application to check for a CYUSB driver
HKR,,DriverBase,,CYUSB.sys
HKR,"Parameters","MaximumTransferSize",0x10001,4096
HKR,"Parameters","DebugLevel",0x10001,2
HKR,,FriendlyName,,%CYUSB_Description%

[CYUSB.AddService]
DisplayName    = %CYUSB_Description%
ServiceType    = 1                  ;
SERVICE_KERNEL_DRIVER
StartType      = 3                  ;
SERVICE_DEMAND_START
ErrorControl   = 1                  ;
SERVICE_ERROR_NORMAL
ServiceBinary  = %10%\System32\Drivers\CYUSB.sys
AddReg         = CYUSB.AddReg
LoadOrderGroup = Base

[CYUSB.Files.Ext]
CYUSB.sys

[CYUSB.AddReg.Guid]
HKR,,DriverGUID,,%CYUSB.GUID%

[Strings]
CYUSB_Provider    = "Cypress"
CYUSB_Company     = "Cypress Semiconductor Corporation"
CYUSB_Description = "Cypress Generic USB Driver"
CYUSB_DisplayName = "Cypress USB Generic"
CYUSB_Install     = "Cypress CYUSB Driver Installation
Disk"
VID_04B4&PID_E175.DeviceDesc="Cypress AN56377 Project 1
Device Driver (3.4.6.000)"
VID_04B4&PID_E176.DeviceDesc="Cypress AN56377 Project 2
Device Driver (3.4.6.000)"
VID_04B4&PID_1003.DeviceDesc="Cypress AN56377 Project 3
Device Driver (3.4.6.000)"
CYUSB.GUID="{AE18AA60-7F6A-11d4-97DD-00010229B959}"
CYUSB_Unused      = "."
```

Appendix B: main.c for Project 1

```
#include <device.h>

/* Create Defines for various Endpoints */
#define EP1 0x01
#define EP2 0x02
#define EP3 0x03
#define EP4 0x04
#define EP5 0x05
#define EP6 0x06

#define ZeroBandwidthInterface 0
#define UnassignedAddress 0

void Interrupt_Transfer (void);
void Bulk_Transfer (void);
void Isochronous_Transfer (void);

extern uint8 USBFS_deviceAddress; /*External variable that contains device address */

uint8 Interrupt_OUT_Buffer[8], EP2_Count;
uint8 Bulk_OUT_Buffer[8], EP4_Count;
uint8 Isochronous_OUT_Buffer[8], EP6_Count ;
uint8 AltSettingNumber;


void main()
{
     CYGlobalIntEnable; /* Enable global interrupts */
     USBFS_Start(0, USBFS_DWR_VDDD_OPERATION);  /* Start USBFS Component based on power settings in DWR */
     while(USBFS_deviceAddress == UnassignedAddress); /* Wait until device address is assigned */

    for(;;)
    {
     if(USBFS_GetConfiguration() != 0)
         {
                 if(USBFS_IsConfigurationChanged() != ZeroBandwidthInterface) /* Check USB configuration has changed */
                 {
                         /* Get the active interface number and reenable OUT EP based on Alternative Setting */
                         AltSettingNumber = USBFS_GetInterfaceSetting(0);
                         if(AltSettingNumber == 0x01) USBFS_EnableOutEP(2);
                         if(AltSettingNumber == 0x02) USBFS_EnableOutEP(4);
                         if(AltSettingNumber == 0x03) USBFS_EnableOutEP(6);
                 }


                 /* Check the Alternative Setting variable to see which endpoint buffers to read and load */
                 switch(AltSettingNumber){

                 case 0x01:Interrupt_Transfer();
                                 break;

                 case 0x02:Bulk_Transfer();
                                 break;

                 case 0x03:Isochronous_Transfer();
                                 break;

                 default:  break; }
     }
     }
}


void Interrupt_Transfer (void)
{
     /* Check to see if OUT Endpoint has data. If so, read EP2 buffer and load back into EP1 for loopback */
     if(USBFS_GetEPState(EP2) == USBFS_OUT_BUFFER_FULL)
     {
                 EP2_Count = USBFS_GetEPCount(EP2);
                 USBFS_ReadOutEP(EP2, Interrupt_OUT_Buffer, EP2_Count);
                 USBFS_LoadInEP(EP1, Interrupt_OUT_Buffer, 8);
                 USBFS_EnableOutEP(EP2);
     }
}

void Bulk_Transfer (void)
{
```

```
    /* Check to see if OUT Endpoint has data. If so, read EP4 buffer and load back into EP3 for loopback */
    if(USBFS_bGetEPState(EP4) == USBFS_OUT_BUFFER_FULL)
    {
                EP4_Count = USBFS_wGetEPCount(EP4);
                USBFS_ReadOutEP(EP4, Bulk_OUT_Buffer, EP4_Count);
                USBFS_LoadInEP(EP3, Bulk_OUT_Buffer, EP4_Count);
                USBFS_EnableOutEP(EP4);
    }
}

void Isochronous_Transfer (void)
{
    /* Check to see if OUT Endpoint has data. If so, read EP6 buffer and load back into EP5 for loopback */
    if(USBFS_bGetEPState(EP6) == USBFS_OUT_BUFFER_FULL)
    {
                EP6_Count = USBFS_wGetEPCount(EP6);
                USBFS_ReadOutEP(EP6, Isochronous_OUT_Buffer, EP6_Count);

                USBFS_LoadInEP(EP5, Isochronous_OUT_Buffer, EP6_Count);
                USBFS_EnableOutEP(EP6);
    }
}
/* [] END OF FILE */
```

Appendix C: USBFS_1_vnd.c for Project 2

Add the following custom declarations to the VENDOR_SPECIFIC_DECLARATIONS section:

```
/***************************************
* Vendor Specific Declarations
***************************************/

/* `#START VENDOR_SPECIFIC_DECLARATIONS` Place your declaration here */
#include "device.h"
#define SET_LED 0xA1
#define CLEAR_LED 0xB1
#define CONTROL_READ 0xA2
#define CONTROL_WRITE 0xB2

uint16 wValue, wIndex;
uint8 ControlDataArray[8];
uint8 USBFS_1_InitNoDataControlTransfer(void);
/* `#END` */
```

Add the following custom vendor command code to the USBFS_1_HandleVendorRqst() function:

```
   /* `#START VENDOR_SPECIFIC_CODE` Place your vendor specific request here */

        wValue = CY_GET_REG16(USBFS_1_wValue);
        wIndex = CY_GET_REG16(USBFS_1_wIndex);

        switch (CY_GET_REG8(USBFS_1_bRequest))
        {
                case SET_LED:

                if(wValue == 0x0001)
                {
                        LED_1_Write(1);
                }
                else if (wValue == 0x0002)
                {
                        LED_2_Write(1);
                }
                else
                {
                }
                        requestHandled = USBFS_1_InitNoDataControlTransfer();
                break;

                case CLEAR_LED:
                if(wValue == 0x0001)
                {
                        LED_1_Write(0);
                }
                else if (wValue == 0x0002)
                {
                        LED_2_Write(0);
                }
                else
                {
                }

                requestHandled = USBFS_1_InitNoDataControlTransfer();
                break;


                case CONTROL_READ:
                USBFS_1_currentTD.wCount = 8;
                USBFS_1_currentTD.pData = ControlDataArray;
                requestHandled = USBFS_1_InitControlRead();
                break;

                case CONTROL_WRITE:
                USBFS_1_currentTD.wCount = 8;
                USBFS_1_currentTD.pData = ControlDataArray;
                requestHandled = USBFS_1_InitControlWrite();
                break;

        }
    /* `#END` */
```

Appendix D: main.c for Project 2

```
#include <device.h>

#define UnassignedAddress 0

extern uint8 USBFS_1_deviceAddress;

uint8 i;

void main()
{
        CYGlobalIntEnable;
        USBFS_1_Start(0, USBFS_1_DWR_VDDD_OPERATION);
        while(USBFS_1_deviceAddress == UnassignedAddress);

        for(;;)
    {

    }
}

/* [] END OF FILE */
```

Appendix E: main.c for Project 3

```
#include <device.h>

#define EP1 0x01
#define ZeroBandwidthInterface 0
#define UnassignedAddress 0

extern uint8 USBFS_deviceAddress;
uint8 AltSettingNumber = 0;
uint16 i = 0;
uint8 EP1_RAM[1023];


void main()
{
        CYGlobalIntEnable;
        USBFS_Start(0, USBFS_DWR_VDDD_OPERATION);
        while(USBFS_deviceAddress == UnassignedAddress);
        for(i=0;i<sizeof(EP1_RAM);i++) EP1_RAM[i] = i;

    for(;;)
    {
                /* Wait for Configuration to Complete */
                if(USBFS_GetConfiguration() != 0)
                {
                        /* Reinitialize after SET_CONFIGURATION or SET_INTERFACE Requests */
                        if(USBFS_IsConfigurationChanged() != 0x00)
                        {
                                /* Get the active interface number and reinitalize DMA */
                                AltSettingNumber = USBFS_GetInterfaceSetting(0);
                                USBFS_LoadInEP(EP1, &EP1_RAM[0], sizeof(EP1_RAM));
                        }

                        /* Check that ISO Interface is active (Not Interface 0) */
                        if(AltSettingNumber != ZeroBandwidthInterface)
                        {
                                /* Check that all data has been transfered and IN Buffer is empty */
                                if (USBFS_GetEPState(1) == USBFS_IN_BUFFER_EMPTY)
                {
                /*Rearm the IN Endpoint (EP1) */
                USBFS_LoadInEP(EP1, USBFS_NULL, sizeof(EP1_RAM));
                        }
                        }
                }
        }
}
/* [] END OF FILE */
```

Document History

Document Title: PSoC® 3 and PSoC 5 USB Transfer Types – AN56377

Document Number: 001-56377

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2811300	BHA/HRID	11/18/2009	New application note
*A	3010626	HRID	08/19/2010	Updated with PSoC 5
*B	3092283	HRID	11/23/2010	The Code Font size has been changed in page 6, 7, 8,9,10 and 11.
*C	3221695	RLRM	04/11/2011	The application note has been extensively updated. The projects have also been updated. The project from AN57294 is also merged into this application note.
*D	3250514	RLRM	05/06/2011	Updated Figure 1, 2, 3, 4, and 23
*E	3281366	RLRM	06/13/2011	Broken links fixed and document formatted as per template.
*F	3451232	RLRM	11/30/2011	Updated content throughout the application note based on software updates. Updated template.
*G	3470131	RLRM	12/20/2011	Updated software version from “PSoC Creator 1.0” to “PSoC Creator 2.0”.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC Solutions

psoc.cypress.com/solutions
PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

PSoC is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSoC Designer, and PSoC Express are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2006–2011. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

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

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

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