

USB 101: An Introduction to Universal Serial Bus 2.0

Author: Robert Murphy

Associated Part Family: PSoC® 1, PSoC 3, PSoC 5LP, PSoC 4200L, USB controllers

Associated Code Examples: See Related Resources

Related Application Notes: See Related Resources

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

Contents

| | | | |
|--------------------------------------------------|----|----------------------------------------------------|----|
| Contents..... | 1 | 11 USB Class Devices | 31 |
| 1 Introduction..... | 2 | 12 USB Enumeration and Configuration | 33 |
| 2 USB History | 2 | 12.1 Dynamic Detection..... | 33 |
| 3 USB Overview | 3 | 12.2 Enumeration | 33 |
| 4 USB Architecture | 4 | 12.3 Configuration | 34 |
| 5 Physical Interface | 7 | 13 Debugging USB Designs..... | 34 |
| 6 USB Speeds | 11 | 13.1 Debugging on the Host Side..... | 34 |
| 7 USB Power | 12 | 13.2 Debugging the Communication | 36 |
| 8 USB Endpoints | 15 | 13.3 Debugging on the Device Side | 40 |
| 9 Communication Protocol | 16 | 14 Acquiring a VID and PID..... | 40 |
| 9.1 Packet Types | 18 | 15 Compliance Testing..... | 40 |
| 9.2 Transaction Types | 21 | 15.1 USB-IF Compliance Testing | 40 |
| 10 USB Descriptors | 25 | 15.2 Microsoft Hardware Certification Testing..... | 47 |
| 10.1 Device Descriptor | 25 | 16 Summary | 48 |
| 10.2 Configuration Descriptor | 26 | 17 Related Resources..... | 49 |
| 10.3 Interface Association Descriptor (IAD)..... | 27 | A Appendix A | 50 |
| 10.4 Interface Descriptor | 27 | A.1 Example PSoC 3 Full-Speed | |
| 10.5 Endpoint Descriptor | 28 | USB Device Descriptors | 50 |
| 10.6 String Descriptor | 29 | B Appendix B | 52 |
| 10.7 Other Miscellaneous Descriptor Types | 29 | B.1 Bus Analyzer Capture of | |
| 10.8 Using Multiple USB Descriptors..... | 30 | USB Enumeration (Example)..... | 52 |

1 Introduction

USB is an interface that connects a device to a computer. With this connection, the computer sends or retrieves data from the device. USB gives developers a standard interface to use in many different types of applications. A USB device is easy to connect and use because of a systematic design process. This application note is intended to help make that process simpler.

The following concepts are covered in this application note:

- USB History
- USB Architecture
- USB Physical Interface
- USB Speeds
- USB Power
- USB Endpoints
- USB Communication Protocol
- USB Descriptors
- USB Class Devices
- USB Enumeration and Configuration Process
- USB Compliance and Windows Logo Testing

Note that USB 3.0 is not discussed in this application note. See the application notes listed in the [Related Resources](#) section for USB 3.0 related application notes. This application note also does not provide any code example for USB 2.0 devices. Refer to the code examples listed in the [Related Resources](#) section for links to code examples for each of the product families.

2 USB History

USB is an industry standard developed for the connection of electronic peripherals such as keyboard, mice, modems, and hard drives to a computer. This standard was developed in order to replace larger and slower connections such as serial and parallel ports. The standard was developed through a joint effort, starting in 1994, between Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. The goals were to develop a single interface that could be used across multiple devices, eliminate the many different connectors currently available at the time, and increase the data throughput of electronic devices.

Over the years, the USB specification has undergone multiple revisions. It all started with USB 1.0, which was finalized in January of 1996. The original specification only included support for two speeds: Low-Speed (LS), which supported 1.5 Mb/s and Full-Speed (FS), which supported 12 Mb/s. While Low-Speed was slower than Full-Speed, it was less susceptible to electromagnetic interference (EMI), which made it attractive to many USB device developers because lower cost components could be used. In 1998, USB 1.1 was developed and added some clarifications and improvements to the USB 1.0 specification. It was not until the release of USB 2.0 in April 2000 that the next major change occurred. This revision added a new speed, High-Speed (HS), to the specification making it capable of 480 Mb/s. This specification revision is backward-compatible with USB 1.1 and 1.0. That same backward-compatibility was maintained when USB 3.0 was announced in November 2008, providing speeds up to 5 Gb/s. With USB 3.0 came a new physical connector as well. More recently, plans for USB 3.1 were announced by USB-IF, which will increase speeds up to 10 Gb/s. USB is currently regulated by the [USB Implementers Forum](#) (USB-IF), which is a nonprofit organization that maintains the USB documents and compliance programs.

3 USB Overview

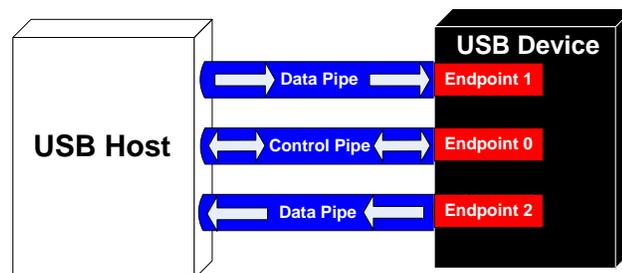
USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller and the root hub. The host controller is a hardware chipset with a software driver layer that is responsible for these tasks:

- Detect attachment and removal of USB devices
- Manage data flow between host and devices
- Provide and manage power to attached devices
- Monitor activity on the bus

At least one host controller is present in a host and it is possible to have more than one host controller. Each controller allows connection of up to 127 devices with the use of external USB hubs. The root hub is an internal hub that connects to the host controller(s) and acts as the first interface layer to the USB in a system. Currently on your PC, there are multiple USB ports. These ports are part of the root hub in your PC. For simplicity, look at the root hub and host controller from the abstract view of a “black box” that we call the host.

USB devices consist of one or more device functions, such as a mouse, keyboard, or audio device for example. Each device is given an address by the host, which is used in the data communication between that device and the host. USB device communication is done through pipes. These pipes are a connection pathway from the host controller to an addressable buffer called an endpoint. An endpoint stores received data from the host and holds the data that is waiting to transmit to the host. A USB device can have multiple endpoints and each endpoint has a pipe associated with it. This is shown in [Figure 1](#).

Figure 1. USB Pipe Model



There are two types of pipes in a USB system, control pipes and data pipes. The USB specification defines four different data transfer types. Which pipe is used depends on the data transfer type.

- **Control Transfers** Used for sending commands to the device, make inquiries, and configure the device. This transfer uses the control pipe.
- **Interrupt Transfers** Used for sending small amounts of bursty data that requires a guaranteed minimum latency. This transfer uses a data pipe.
- **Bulk Transfers** Used for large data transfers that use all available USB bandwidth with no guarantee on transfer speed or latency. This transfer uses a data pipe.
- **Isochronous Transfers** Used for data that requires a guaranteed data delivery rate. Isochronous transfers are capable of this guaranteed delivery time due to their guaranteed latency, guaranteed bus bandwidth, and lack of error correction. Without the error correction, there is no halt in transmission while packets containing errors are resent. This transfer uses a data pipe.

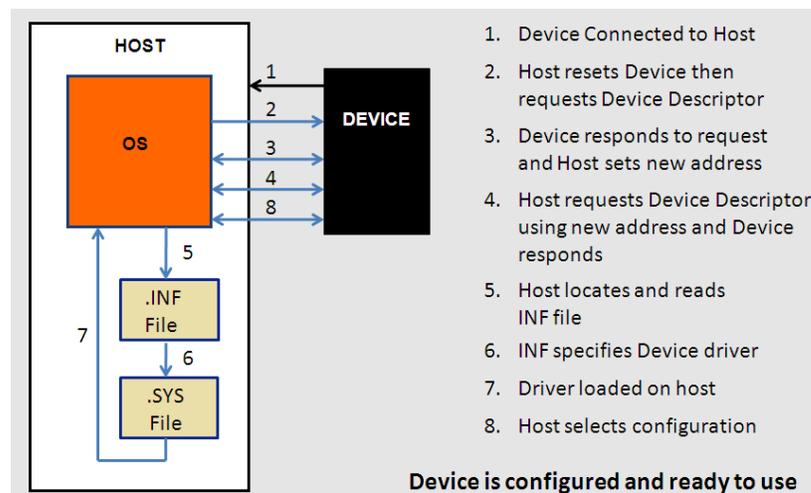
Every device has a control pipe and it is through this pipe that control transfers to send and receive messages from the device are performed. Optionally, a device may have data pipes for transferring data through interrupt, bulk, or isochronous transfers. The control pipe is the only bidirectional pipe in the USB system. All the data pipes are unidirectional.

Each endpoint is accessed with a device address (assigned by the host) and an endpoint number (assigned by the device). When information is sent to the device, the device address and endpoint number are identified with a token packet (discussed later in [Communication Protocol](#) section). The host initiates this token packet before a data transaction.

When a USB device is first connected to a host, the USB enumeration process is initiated. Enumeration is the process of exchanging information between the device and the host that includes learning about the device. Additionally, enumeration includes assigning an address to the device, reading descriptors (which are data structures that provide information about the device), and assigning and loading a device driver. This entire process can occur in seconds. For more information see the [USB Enumeration and Configuration](#) section. When this process is complete, the device is ready to transfer data to the host. The flow chart of the general enumeration process is shown in [Figure 2](#). Two files are affiliated with enumeration and the loading of a driver. They exist on the host side.

- **.INF** – A text file that contains all the information necessary to install a device, such as driver names and locations, Windows registry information, and driver version information.
- **.SYS** – The driver needed to communicate effectively with the USB device.

Figure 2. Sequence of Enumeration Events



After a device is enumerated, the host directs all traffic flow to the devices on the bus. Because of this, no device can transfer data without a request from the host controller.

4 USB Architecture

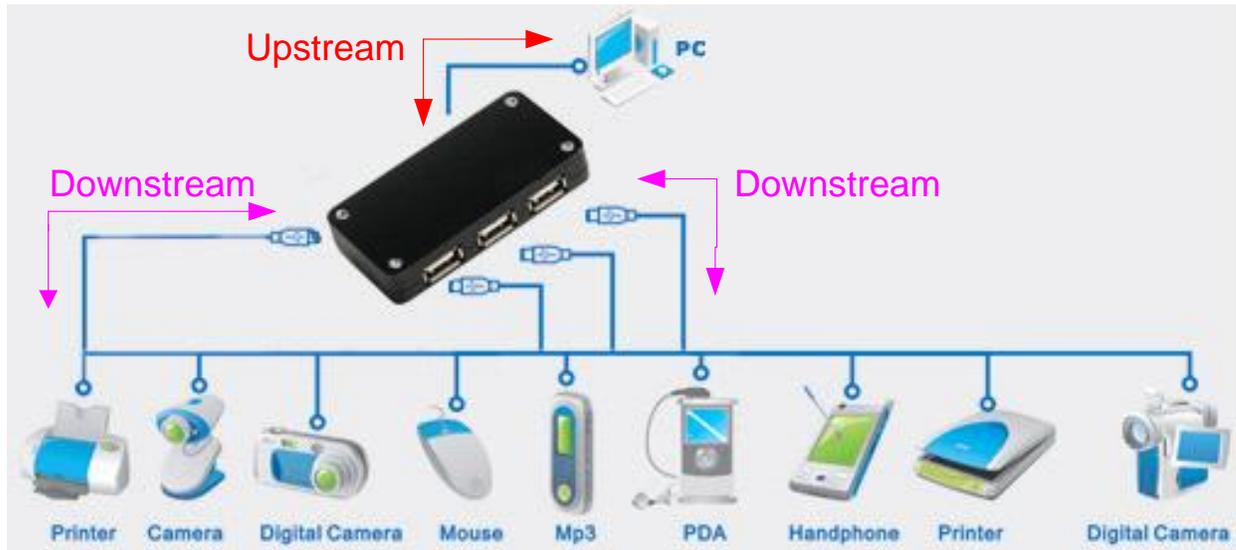
Only one host can exist in the system and communication with devices is from the host's perspective. A host is an "upstream" component, while a device is a "downstream" component; [Figure 3](#) shows a representation of this. Data moved from the host to the peripheral is an OUT transfer. Data moved to the host from the peripheral is an IN transfer. The host, specifically the host controller, controls all traffic and issues commands to devices. There are three common types of USB host controllers:

Universal Host Controller Interface (UHCI): Produced by Intel for USB 1.0 and USB 1.1. Using UHCI requires a license from Intel. This controller supports both Low-Speed and Full-Speed.

Open Host Controller Interface (OHCI): Produced for USB 1.0 and 1.1 by Compaq, Microsoft, and National Semiconductor. Supports Low-Speed and Full-Speed and tends to be more efficient than UHCI by performing more functionality in hardware.

Extended Host Controller Interface (EHCI): Created for USB 2.0 after USB-IF requested that a single host controller specification be created. EHCI is used for High-Speed transactions and delegates Low-Speed and Full-Speed transactions to an OHCI or UHCI sister controller.

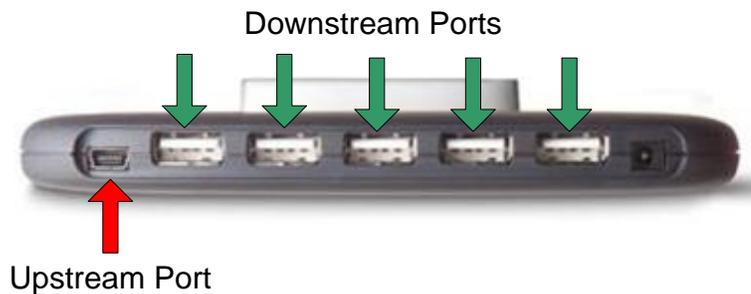
Figure 3. Many Peripherals Can Connect to One Host



One or more devices are attached to a host. Each device has an address and responds to host commands that are addressed to it. Devices are expected to have some form of functionality and not simply be passive. Devices contain one upstream port. Ports are the physical USB connection point on the device.

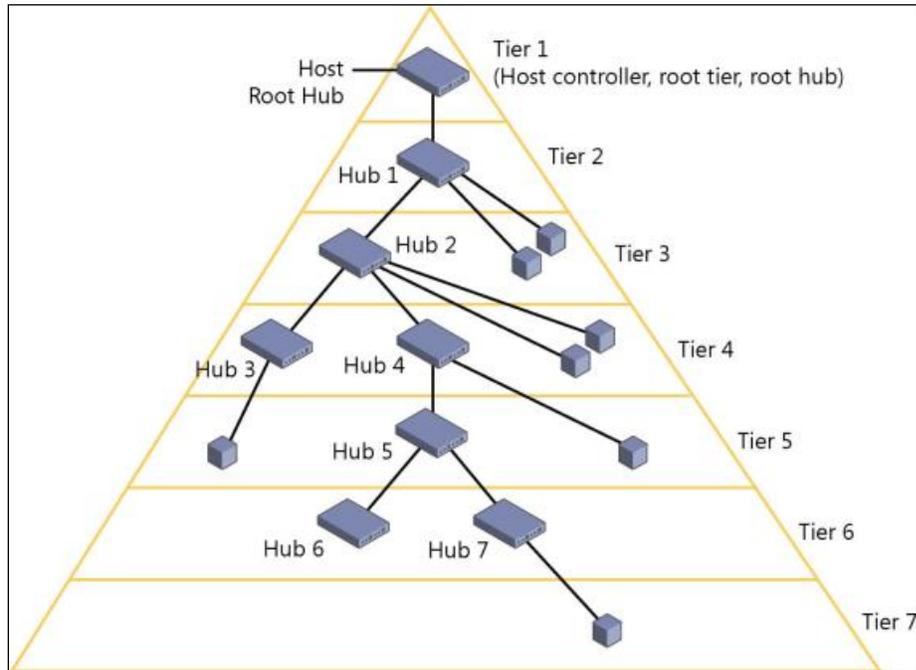
A hub is a specialized device that allows the host to communicate with multiple peripheral devices on the bus. Unlike USB peripheral devices, such as a mouse that has actual functionality, a hub device is transparent and is intended to act as a pass-through. A hub also acts as a channel between the host and the device. Hubs have additional attachment points to allow the connection of multiple devices to a single host. A hub repeats traffic to and from downstream devices through one upstream port and up to seven downstream ports. The hub, however, does not have any host capabilities.

Figure 4. Hub Connections



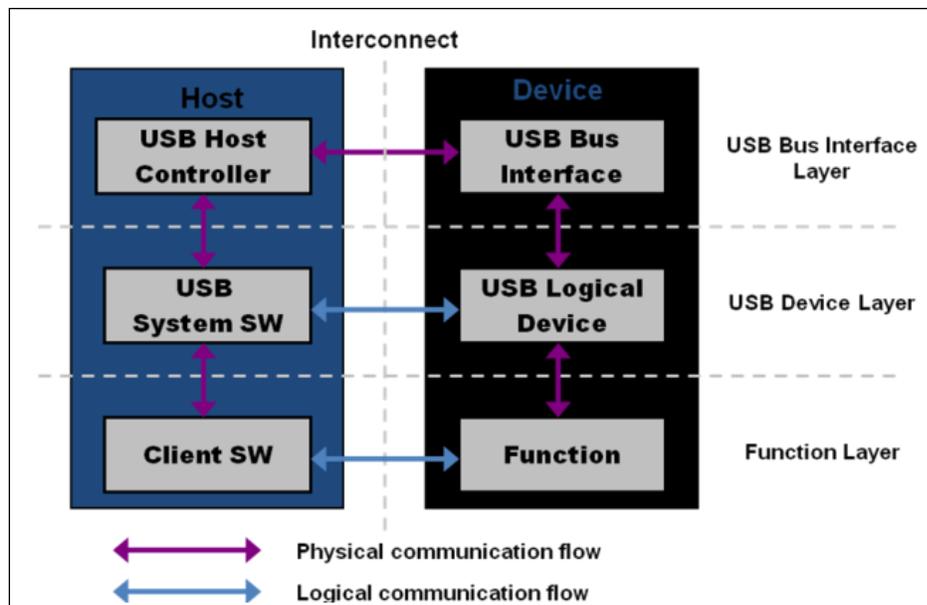
As mentioned earlier, up to 127 devices can be connected to the host controller with the use of hubs. This limitation is based on the USB protocol, which limits the device address to 7 bits. Additionally, a maximum of five hubs can be chained together, which is limited due to timing constraints of hub and cable propagation delays. Figure 5 shows a diagram of the USB tier system that represents the limitation of chaining hubs and devices together. You can see that with the limitation on chaining hubs together, this produces a seven tiered system.

Figure 5. USB Connection Tier



Another way to look at the USB interface is to divide it into different layers, as shown in Figure 6. The Bus Interface Layer provides the physical connection, electrical signaling, and packet connectivity. This is the layer that is handled by the hardware in a device. This is accomplished with the physical interface external to the device. The Device Layer is the view the USB system software has for performing USB operations such as sending and receiving information. This is accomplished with a Serial Interface Engine, which is also internal to the device. Finally, the Function Layer is the software side of things. This is the portion of a USB device that does something with the information it received or does something to gather data to transfer to the host. Figure 6 shows this abstraction.

Figure 6. Interface Abstraction

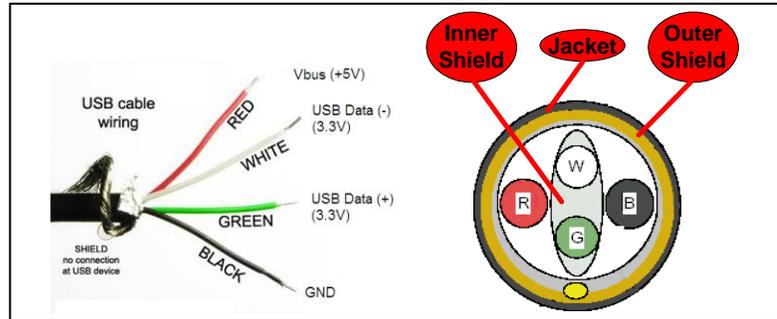


5 Physical Interface

From a high-level overview, the physical interface of USB has two components: cables and connectors. These connectors connect devices to a host.

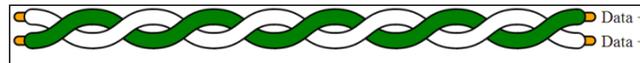
A USB cable consists of multiple components that are protected by an insulating jacket. Underneath the jacket is an outer shield that contains a copper braid. Inside the outer shield are multiple wires: a copper drain wire, a V_{BUS} wire (red), and a ground wire (black). An inner shield made of aluminum contains a twisted pair of data wires as seen in Figure 7. There is a D+ wire (green) and a D- wire (white).

Figure 7. Inside a USB Cable



In Full-Speed and High-Speed devices, the maximum cable length is 5 meters. To increase the distance between the host and a device, you must use a series of hubs and 5-meter cables. While USB extension cables exist in the market, using them to exceed 5 meters is against the USB specification. Low-Speed devices have slightly different specifications. Their cable length is limited to 3 meters and Low-Speed cables are not required to be a twisted pair as Figure 8 shows.

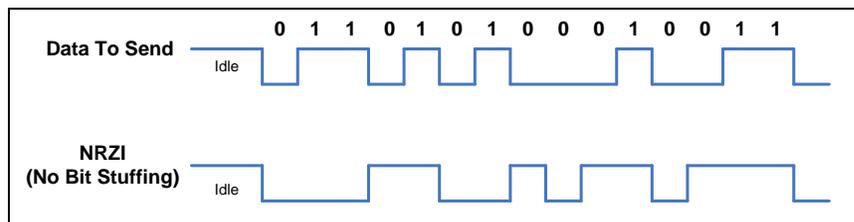
Figure 8. USB Twisted Pair Data Wires



The V_{BUS} wire gives a constant 4.40 - 5.25 V supply to all attached devices. While USB supplies up to 5.25 V to devices, the data lines (D+ and D-) function at 3.3 V. The USB interface uses a differential transmission that is non-return-to-zero inverted (NRZI) encoded with bit stuffing across the twisted pair.

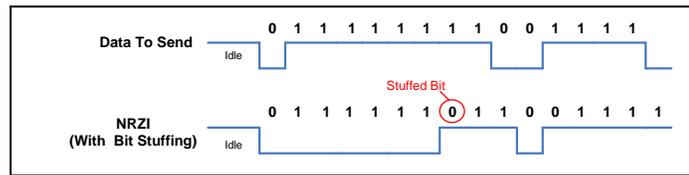
NRZI encoding is a method for mapping a binary signal for transmission over some medium, in this case, a USB cable. With this encoding scheme, a logic 1 is represented by no change in voltage level and a logic 0 is represented by a change in voltage level as Figure 9 shows. On the top is the data that will be transmitted over USB. On the bottom is the encoded NRZI data.

Figure 9. Data to NRZI Encoding



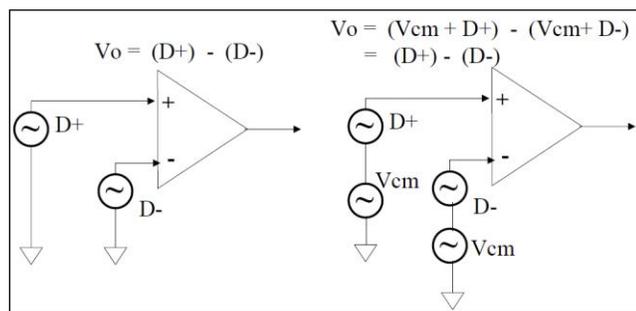
The bit stuffing occurs by inserting a logic 0 following seven consecutive logic 1s. The purpose of the bit stuffing is for synchronization of the USB hardware by maintaining phase-locked loop (PLL). If there are too many logic 1s in the data, then there may not be enough transitions in the NRZI encoded stream to synchronize from. The receiver on the USB hardware automatically detects this extra bit and disregards it. This extra bit stuffing contributes to the extra overhead on the USB. Figure 10 shows an example of NRZI data with bit stuffing. Notice in the "Data to Send" stream there are eight 1s. In the encoded data, after the sixth logic 1, a logic 0 is inserted. The seventh and eighth logic 1 then follow this logic 0.

Figure 10. Data to NRZI Encoding with Bit Stuffing



The hardware in USB devices will handle all the encoding and bit stuffing upon receiving any data and before transmitting any data. The reason for using the differential D+ and D- signal is for rejecting common-mode noise. If noise becomes coupled into the cable, it will normally be present on all wires in the cable. With the use of a differential amplifier in the USB hardware internal to the host and device, the common-mode noise can be rejected as shown in Figure 11.

Figure 11. USB Input Differential Amplifier Buffer



USB communication occurs through many different signaling states on the D+ and D- lines. Some of these states transmit the data while others are used as specific signaling conditions. These states are described below with a quick reference list located in Table 1.

Differential 0 and Differential 1: These two states are used for general data communication across USB. Differential 1 is when the D+ line is high and the D- line is low. Differential 0 is when the D+ line is low and the D- line is high. An example of USB data communication is shown in Figure 12.

J-State and K-State: In addition to the differential signals, the USB specification defines two additional differential states: J-State and K-State. Their definitions depend on the device speed. On a Full-Speed and High-Speed device, a J-State is a Differential 1 and a K-State is a Differential 0. The opposite is true for a Low-Speed device.

Single Ended Zero (SE0): Condition that occurs when both D+ and D- are driven low. This condition indicates a reset, disconnect, or End of Packet.

Single Ended One (SE1): Condition that occurs when D+ and D- are both driven high. This condition does not ever occur intentionally and should not be seen occurring in a USB design.

Idle: Condition that occurs before and after a packet is sent. An Idle condition is signified by one of the data lines being low and the other line being high. The definition of high versus low depends on device speed. On a Full-Speed device, an idle condition consists of D+ being high and D- being low. The opposite is true for a Low-Speed device.

Resume: Used to wake a device from a suspend state. This is done by issuing a K-State.

Start of Packet (SOP): Occurs before the start of any Low-Speed or Full-Speed packet when the D+ and D- lines transition from an idle state to a K-State.

End of Packet (EOP): Occurs at the end of any Low-Speed or Full-Speed packet. An EOP occurs when an SE0 state occurs for 2 bit times (bit times are discussed later), followed by a J-State for 1 bit time.

Reset: Occurs when an SE0 state lasts for 10 ms. After a SE0 has occurred for at least 2.5 ms, the device may recognize the reset and begin to enter a reset state.

Keep Alive: Signal used in Low-Speed devices. Low-Speed devices lack a Start-of-Frame packet that is required to prevent suspend. They use an EOP every 1 ms to keep the device from entering suspend.

Figure 12. USB D+ and D- Communication

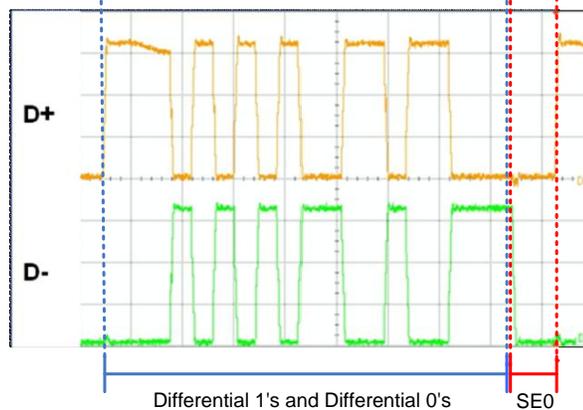


Table 1. USB Communication States

| Bus State | Indication |
|-----------------------|--------------------------------------------------------|
| Differential 1 | D+ High, D- Low |
| Differential 0 | D+ Low, D- High |
| Single Ended 0 (SE0) | D+ and D- Low |
| Single Ended 1 (SE1) | D+ and D- High |
| J-State: | |
| Low-Speed | Differential 0 |
| Full-Speed | Differential 1 |
| High-Speed | Differential 1 |
| K-State: | |
| Low-Speed | Differential 1 |
| Full-Speed | Differential 0 |
| High-Speed | Differential 0 |
| Resume State: | K-State |
| Start of Packet (SOP) | Data lines switch from idle to K-State. |
| End of Packet (EOP) | SE0 for 2 bit time followed by J-State for 1 bit time. |

Figure 13 and Figure 14 show the different USB ports and connectors available. The upstream connection always uses a Type A port and connector, while the device uses Type B ports and connectors. Initially, the USB specification included only the larger Type A and Type B connectors for devices but later included the Mini and Micro connections. These Mini and Micro connectors were initially developed for USB On-the-Go (USB OTG), which is a USB specification that allows devices that would normally act as slaves to become hosts. This is why Figure 14 shows the Mini and Micro ports as Mini-AB and Micro-AB. However, due to the smaller size of the Mini-B and Micro-B connectors compared to Type B, they were adopted in many electronics despite lacking USB OTG capabilities.

Figure 13. USB Connector Size Comparison

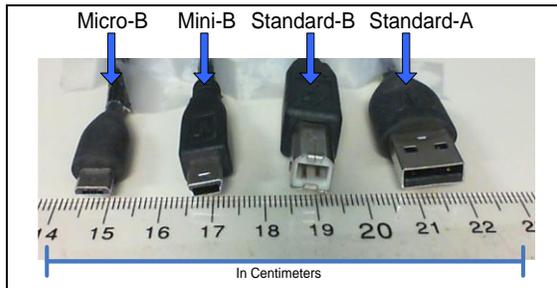


Figure 14. USB Ports and Connectors

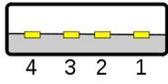
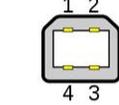
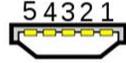
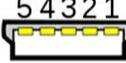
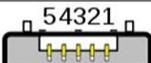
| Type | Port Image | Connector Image |
|----------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| Type A |  |  |
| Type B |  |  |
| Mini-AB |  |  |
| Mini-B |  |  |
| Micro-AB |  |  |
| Micro-B |  |  |

Figure 14 shows that the Mini and Micro connectors have five pins, rather than four. The extra pin is the ID pin and identifies the host and the device in OTG applications. Since PSoC does not support USB OTG, this application note does not include details about it. The reason for having different connection types (Type A and Type B) is to prevent loopback connections on hubs. Some USB devices also contain a captive, or attached cable, with the only visible connector being the Type A. Table 2 and Table 3 show the pinout for USB connectors depending on the connector type.

Table 2. USB Standard Connector Pinout

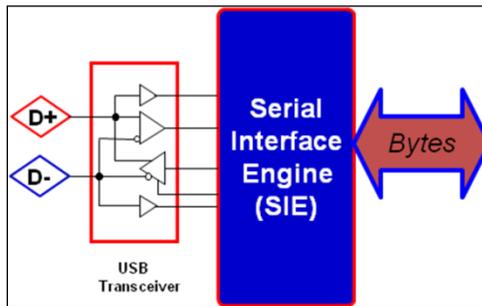
| Pin | Name | Color | Function |
|-----|------------------|-------|----------|
| 1 | V _{BUS} | Red | +5 V |
| 2 | D- | White | Data (-) |
| 3 | D+ | Green | Data (+) |
| 4 | GND | Black | Ground |

Table 3. USB Mini/Micro Connector Pinout

| Pin | Name | Color | Function |
|-----|------------------|-------|-----------------------------------------------------------------------------------------------------------------|
| 1 | V _{BUS} | Red | +5 V |
| 2 | D- | White | Data (-) |
| 3 | D+ | Green | Data (+) |
| 4 | ID | NA | Identifies Type A and Type B Plug: A plug: connected to Signal ground B plug: not connected |
| 5 | GND | Black | Ground |

There are two main hardware blocks required to interface with USB: a transceiver, also known as a PHY (abbreviation for Physical Layer), and a Serial Interface Engine, also known as an SIE. The transceiver provides the hardware interface between the USB connector and the chip circuitry that controls USB communication. The SIE is the core of the USB hardware. It performs many functions such as decoding and encoding the USB data, error correction, bit stuffing, and signaling. SIEs can take many different forms. They are not regulated by the USB specification unlike transceivers. In fact, some devices incorporate SIE that are more software based to reduce cost, while other devices use more of a hardware-centric SIE.

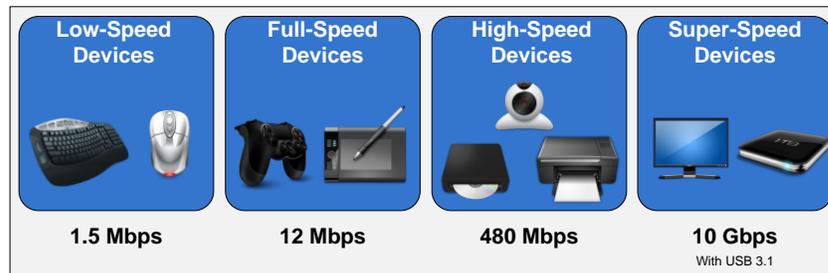
Figure 15. USB Hardware Interface at the Device



6 USB Speeds

Currently, the USB specification defines four speeds for a USB system: Low-Speed, Full-Speed, Hi-Speed, and SuperSpeed. Currently, Cypress only supports Full-Speed on the PSoC family of devices and Low-Speed, Full-Speed, High-Speed, and SuperSpeed on our various dedicated USB devices. As a result, these three speeds will be the focus of this application note.

Figure 16. USB Transfer Speeds



Newer hosts can always communicate with devices of lower speed. For example, a High-Speed Host can communicate with a Low-Speed device, but a Full-Speed Host cannot communicate with a High-Speed device.

Low-, Full-, and High-Speed devices are often advertised as 1.5 Mb/s, 12 Mb/s, and 480 Mb/s, respectively. However, these are bus rates and not data rates. The actual data rates are affected by bus loading, transfer type, overhead, OS, and so forth. The actual limits of the data transfer are.

- Low-Speed devices:
 - Examples: keyboards, mice, and game peripherals
 - Bus Rate: 1.5 Mb/s
 - Maximum Effective Data Rate: 800 B/s
- Full-Speed devices
 - Examples: phones, audio devices, and compressed video
 - Bus Rate: 12 Mb/s
 - Maximum Effective Data Rate: 1.2 MB/s

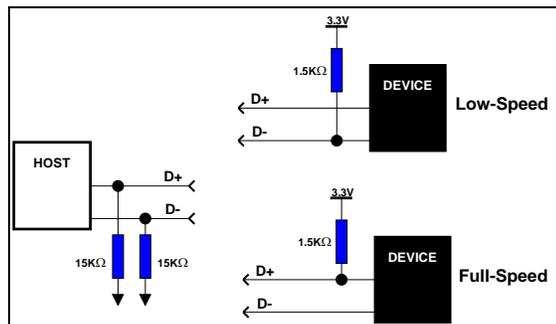
- Hi-speed devices
 - Examples: video, imaging, and storage devices
 - Bus Rate: 480 Mb/s
 - Maximum Effective Data Rate: 53 MB/s

When a USB device is connected to a host, the speed of the device needs to be detected. This is done with pull-up resistors on the D+ or D- line. A 1.5-k Ω pull-up on the D+ line indicates that the attached device is a Full-Speed device. A 1.5-k Ω pull-up resistor on the D- line indicates the attached device is a Low-Speed device. This can be seen in Figure 17.

High-Speed devices start as Full-Speed devices, so they have a 1.5-k Ω pull-up on the D+ line. When the device is connected, it emits a sequence of J-States and K-States during the reset phase of enumeration. If the hub supports High-Speed, then the pull-up resistor is removed.

The pull-up resistor is essential to USB enumeration. Without the pull-up resistor, USB assumes that there is nothing attached to the bus. Some devices require an external pull-up resistor on the D+/D- line. PSoC, however, implements the required pull-up resistor internal to the device, which eliminates the need for this external component.

Figure 17. USB Speed Detection



One common misconception about speed on a USB device is that a device listed as USB 2.0 indicates that the device is High-Speed. All Hi-Speed devices are USB 2.0, but this is because Hi-Speed support was added with USB 2.0. The USB 2.0 specification includes Full- and Low-Speed devices as well.

These speeds also have an effect on the bit timing for USB signaling, such as the End of Packet (EOP) signal. A Low-Speed and Full-Speed USB device will use a 48-MHz clock for the SIE and the other USB clocking purposes. This 48-MHz clock and the bus speed is what will determine USB bit times:

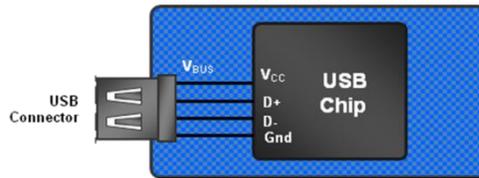
- **Full-Speed:** 48 MHz / 12 Mb/s = 4 clocks per bit time.
- **Low-Speed:** 48 MHz / 1.5 Mb/s = 32 clocks per bit time.

7 USB Power

When it comes to USB power, there are two device categories: bus powered and self powered.

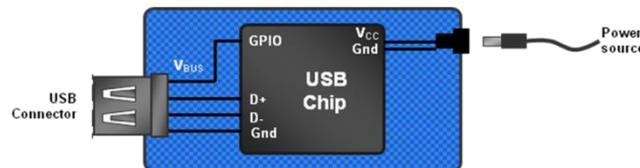
Bus power is one of the many benefits of a USB design. It allows the device to sustain itself without the need for a bulky power supply either internally or externally because the device draws its power from the bus. The power available on the bus is either provided by the host or a hub. When dealing with a bus-powered device, users must consider its power consumption before the device is put into a configured state. This is the time from when the device first connects to the bus to when the host sends the device a SET_CONFIGURATION command after the enumeration steps are complete. Before a device is configured, it must not use more than 100 mA, defined as one unit load in the USB specification, of power for low, full, or high-speed devices. During the configuration phase, the device requests a power budget. There are two classifications for a bus-powered device: high-powered and low-powered devices. A low-powered device draws, at most, 100 mA and a high-powered device draws, at most, 500 mA. Anything over 500 mA requires the device to be self-powered.

Figure 18. USB Bus-powered Device



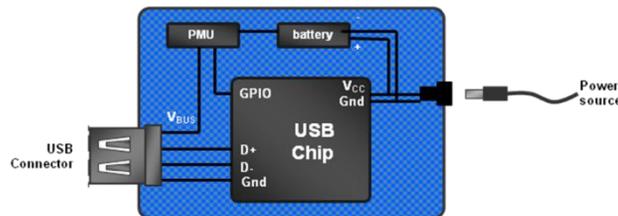
Self-powered devices supply their own power with the use of an external power source such as a DC power adapter or a battery. A self-powered device needs a different set of considerations in a design. The USB specification requires that self-powered devices monitor their V_{BUS} line at all times. A device must remove power from its D+/D- pull-up resistor within a certain time frame of V_{BUS} being removed. The reason for this is to prevent back voltage upstream to the host or hub. Failing to meet this requirement can result in USB compliance testing failure. However, a self-powered hub does have the ability to draw up to 100 mA from the bus.

Figure 19. USB Self-powered Device



Devices can also combine the two power modes and be both a bus-powered and self-powered device. This becomes common when a device runs off a battery. Normally, the device is self-powered, but V_{BUS} is used to charge the battery and provide power to the device while the battery is changing. Technically, this device is a self-powered device and is declared as such in the USB descriptors, but the device requests a power budget from the host. Similar to a self-powered device, monitoring of V_{BUS} is still required in these hybrid designs and the removal of power to the D+/D- pull-up resistors must still occur. In this application, some type of a power management system needs to be implemented to monitor the battery voltage, charging status, and control the switching between the battery power and an external source.

Figure 20. USB Hybrid Powered Device



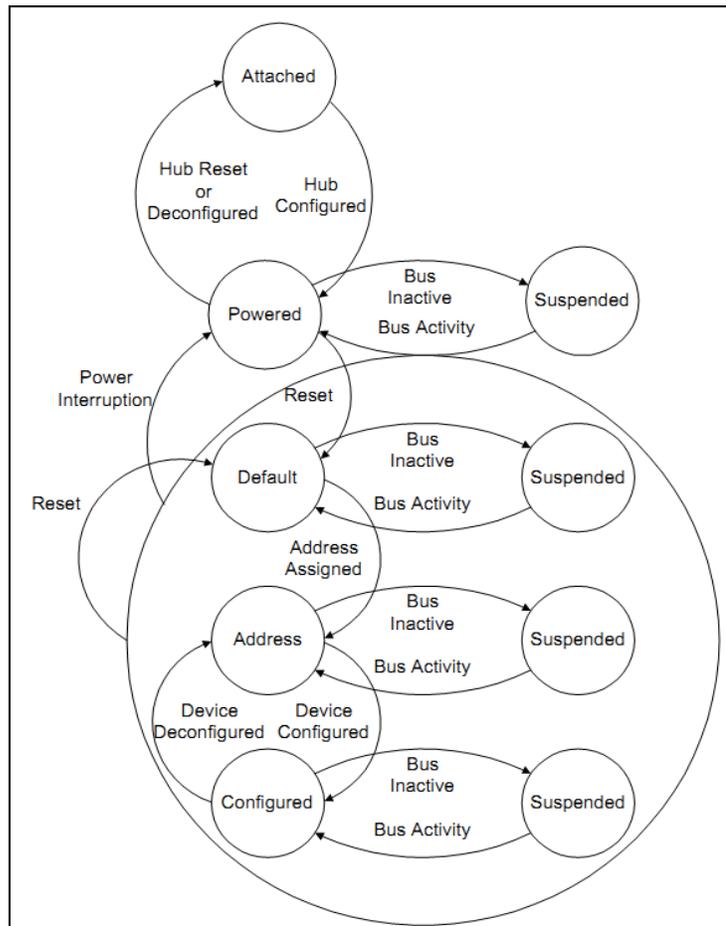
Additionally all USB devices, regardless of how the device is powered, must consider their suspend current. The device's suspend current is the current that is taken from V_{BUS} , when the host is put in suspend mode (also called standby mode). Suspend mode occurs when there is no bus activity for 3 ms. Even when there is no active data transmission, the host uses Start of Frame (SOF) tokens to keep devices out of a suspend state. The exception to this is a Low-Speed device that does not have SOF packets. Low-Speed devices use End of Packet (EOP) transitions, as a Keep Alive Signal every 1 ms when there is no low-speed data on the bus. When the bus becomes inactive, a device must enter suspend and pull no more than 2.5 mA of current. To conform to this requirement, designers must make sure they turn off LEDs and other sinks of power before the device goes into the suspend state. A USB device leaves a suspend state once any activity is detected on the bus. If a device has remote wake-up capability, it can signal resume, then wait to see if the host acknowledges the request rather than waiting for the host to resume activity. For more information on suspend current, see Section 11.4.3 of the USB specification.

Various USB states relate to USB power that a designer needs to know. These states are often seen in USB documentation and apply to the enumeration of a USB device.

- **Attached State:** Occurs when a device is attached to a host/hub, but does not give any power to the V_{BUS} line. This is commonly seen if the hub detects an over current event. The device is still attached, but the hub removes power to it.
- **Powered:** A device is attached to the USB and has been powered, but has not yet received a reset request.
- **Default:** A device is attached to the USB, is powered, and has been reset by the host. At this point, the device does not have a unique device address. The device responds to address 0.
- **Address:** A device is attached to the USB, powered, has been reset, and has had a unique address assigned to it. The device however has not yet been configured.
- **Configured:** The device is attached to the USB, powered, has been reset, assigned a unique address, has been configured, and is not in a suspend state. At this point, bus-powered devices can draw more than 100 mA.
- **Suspend:** As mentioned earlier, occurs when the device is attached and configured, but has not seen activity on the bus for 3 ms.

The USB specification (Figure 9-1 in USB specification) has a diagram that illustrates how these power modes are related and transitioned to. See [Figure 21](#).

Figure 21. Device State Diagram



USB power is communicated in 2 mA units for Low-Speed, Full-Speed, and High-Speed USB devices. For example, a Full-Speed device that must have 100 mA of operational power communicates a value of 50 during enumeration.

When developing a USB design, consider how much power your device consumes from the bus. The root hub gets its power from the supply of the host PC. If the host is attached to AC power, then the USB specification requires the host provide 500 mA of power to each port on the hub. This is what causes the 500 mA limitation on bus-powered devices. If the host PC is battery powered, then it has the option of supplying either 100 mA or 500 mA to each port on the hub. When attaching a device to a hub that is bus-powered, the device must be low power and not consume more than 100 mA. A bus-powered hub has a total of 500 mA to distribute between all attached devices.

8 USB Endpoints

In the USB specification, a device endpoint is a uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device. The [USB Enumeration and Configuration](#) section describes a step in which the device responds to the default address. This occurs before other descriptor information such as the endpoint descriptors are read by the host later in the enumeration process. During this enumeration sequence, a special set of endpoints are used for communication with the device. These special endpoints, collectively known as the Control Endpoint or Endpoint 0, are defined as Endpoint 0 IN and Endpoint 0 OUT. Even though Endpoint 0 IN and Endpoint 0 OUT are two endpoints, they look and act like one endpoint to the developer. Every USB device must support Endpoint 0. For this reason, Endpoint 0 does not require a separate descriptor.

In addition to Endpoint 0, the number of endpoints supported in any particular device is based on its design requirements. A simple design such as a mouse may need only one IN endpoint. More complex designs may need several data endpoints. The USB specification sets a limit on the number of endpoints to 16 for each direction (16 IN/16 OUT – 32 Total) for High-Speed and Full-Speed devices, which does not include the control endpoints 0 IN and 0 OUT. Low-Speed devices are limited to two endpoints. USB Class devices may set a greater limit on the number of endpoints. For example, a Low-Speed human interface device (HID) design may have no more than two data endpoints — typically one IN endpoint and one OUT endpoint. Data endpoints are bidirectional by nature. It is not until they are configured that they take on a single direction (becoming unidirectional). Endpoint 1, for example, can be either an IN or OUT endpoint. It is in the device descriptors that it will officially make Endpoint 1 an IN endpoint.

Endpoints use cyclic redundancy checks (CRCs) to detect errors in transactions. The CRC is a calculated value used for error checking. The actual calculation equation is explained in the USB specification and the handling of these calculations is taken care of by the USB hardware so that the proper response can be issued. The recipient of a transaction checks the CRC against the data. If the two match, then the receiver issues an ACK. If the data and the CRC do not match, then no handshake is sent. This lack of a handshake tells the sender to try again.

The USB specification further defines four types of endpoints and sets the maximum packet size based on both the type and the supported device speed. Developers use the endpoint descriptor to identify the type of endpoint and maximum packet size based on their design requirements. The four types of endpoints and characteristics are:

Control Endpoint – These endpoints support control transfers, which all devices must support. Control transfers send and receive device information across the bus. The advantage of control transfers is guaranteed accuracy. Errors that occur are properly detected and the data is resent. Control transfers have a 10 percent reserved bandwidth on the bus in low and Full-Speed devices (20 percent at High-Speed) and give USB system level control.

Interrupt Endpoints – These endpoints support interrupt transfers. These transfers are used on devices that must use a high reliability method to communicate a small amount of data. This is commonly used in HID designs. The name of this transfer can be misleading. It is not truly an interrupt, but uses a polling rate. However, you get a guarantee that the host checks for data at a predictable interval. Interrupt transfers give guaranteed accuracy as errors are properly detected and transactions are retried at the next transaction. Interrupt transfers have a guaranteed bandwidth of 90 percent on Low- and Full-Speed devices and 80 percent on High-Speed devices. This bandwidth is shared with isochronous endpoints.

Interrupt endpoint maximum packet size is a function of device speed. High-Speed capable devices support a maximum packet size of 1024 bytes. Full-Speed capable devices support a maximum packet size of 64 bytes. Low-Speed devices support a maximum packet size of 8 bytes.

Bulk Endpoints –These endpoints support bulk transfers, which are commonly used on devices that move relatively large amounts of data at highly variable times where the transfers can use any available bandwidth space. They are the most common transfer type for USB devices. Delivery time with a bulk transfer is variable because there is no set aside bandwidth for the transfer. The delivery time varies depending on how much bandwidth on the bus is available, which makes the actual delivery time unpredictable. Bulk transfers give guaranteed accuracy because errors are properly detected and transactions are resent. Bulk transfers are useful in moving large amounts of data that are not time sensitive.

A bulk endpoint maximum packet size is a function of device speed. High-Speed capable devices support a maximum BULK packet size of 512 bytes. Full-Speed capable devices support a maximum packet size of 64-bytes. Low-Speed devices do not support bulk transfer types.

Isochronous Endpoints – These endpoints support isochronous transfers, which are continuous, real-time transfers that have a pre-negotiated bandwidth. Isochronous transfers must support streams of error tolerant data because they do not have an error recovery mechanism or handshaking. Errors are detected through the CRC field, but not corrected. With isochronous, you get the tradeoff of guaranteed delivery versus guaranteed accuracy. Streaming music or video are examples of an application that uses isochronous endpoints because the occasional missed data is ignored by the human ears and eyes. Isochronous transfers have a guaranteed bandwidth of 90 percent on Low- and Full-Speed devices (80 percent on High-Speed devices) that is shared with interrupt endpoints.

High-Speed capable devices support a maximum packet size of 1024 bytes. Full-Speed devices support a maximum packet size of 1023 bytes. Low-Speed devices do not support isochronous transfer types. There are special considerations with isochronous transfers. You generally want 3x buffering to ensure data is ready to go by having one actively transmitting buffer, another buffer loaded and ready to transfer, and a third buffer being actively loaded.

Table 4. Endpoint Transfer Type Features

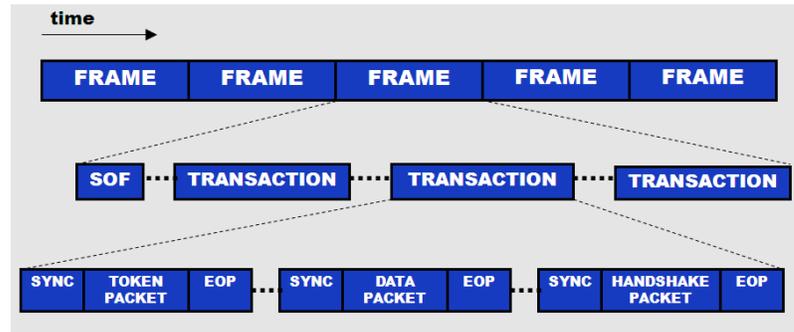
| Transfer Type | Control | Interrupt | Bulk | Isochronous |
|--------------------------|--------------------------------------|--------------------------|--------------------------|------------------------------------|
| Typical Use | Device Initialization and Management | Mouse and Keyboard | Printer 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%) ^[1] | No | Yes (90%) ^[1] |
| Guaranteed Latency | No | Yes | No | Yes |
| Maximum Transfer Size | 64 bytes | 64 bytes | 64 bytes | 1023 bytes (FS) 1024 bytes (HS) |
| 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.

9 Communication Protocol

If you look at the USB communication from a time perspective, it contains a series of frames. Each frame consists of a Start of Frame (SOF) followed by one or more transactions. Each transaction is made up of a series of packets. A packet is preceded with a sync pattern and ends with an End of Packet (EOP) pattern. At a minimum, a transaction has a token packet. Depending on the transaction, there may be one or more data packets and some transactions may or may not have a handshake packet.

Figure 22. USB Communication from a Time Perspective



Transactions are an exchange of packets and are comprised of three different packets; a token packet, optional data packet, and a handshake packet.

Transactions are placed within frames and are never split across frames (with the exception of High-Speed isochronous transfers) or interrupt the middle of another transaction. Figure 23 shows a transaction block diagram.

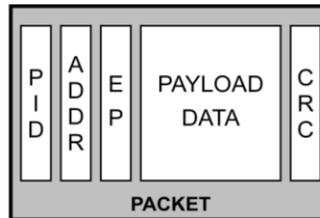
Figure 23. Transaction Block Diagram



Each packet can contain different pieces of information. What information is included depends on the packet type. The following is a list of the potential information that can be included with a packet and Figure 24 shows the potential composition of a packet. Think of Figure 24 as a packet template; information can be added and taken out as needed.

- **Packet ID (PID)** – (8 bits: 4 type bits and 4 error check bits). These bits declare a transaction as an IN/OUT/SETUP/SOF.
- **Optional Device Address** – (7 bits: Max of 127 devices)
- **Optional Endpoint Address** – (4 bits: Max of 16 endpoints). The USB specification supports up to 32 endpoints. While 4 bits gives a maximum value of 16, we achieve 32 endpoints with an IN PID and an endpoint address between 1 and 16 and an OUT PID with an endpoint address between 1 and 16, giving a total of 32. Remember that this is the endpoint address, not the endpoint number.
- **Optional Payload Data** – (0 to 1023 bytes)
- **Optional CRC** (5 or 16 bits)

Figure 24. USB Packet Contents



9.1 Packet Types

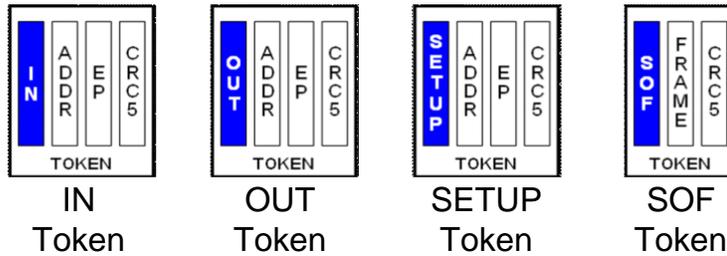
Figure 24 can potentially represent four packet types.

- Token packets
 - Initiate transaction
 - Identify device involved in transaction
 - Always sourced by the host
- Data packets
 - Delivers payload data
 - Sourced by host or device
- Handshake packets
 - Acknowledge error-free data receipt
 - Sourced by receiver of data
- Special packets
 - Facilitates speed differentials
 - Sourced by host-to-hub devices

As mentioned earlier, everything in the packet with the exception of the PID is optional. Token, data, and handshake packets have different combinations of the packet information. Which information is included is identified in the token packet, data packet, and handshake packet sections.

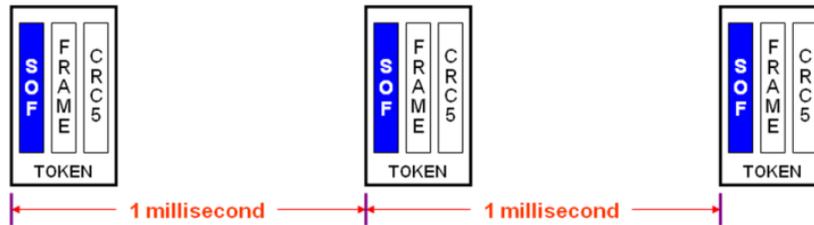
Token Packets: Token packets always come from the host and are used to direct traffic on the bus. The function of the token packet depends on the activity performed. IN tokens are used to request that devices send data to the host. OUT tokens are used to precede data from the host. SETUP tokens are used to precede commands from the host. SOF tokens are used to mark time frames. With an IN, OUT, and SETUP token packet, there is a 7-bit device address, 4-bit endpoint ID, and 5-bit CRC. Figure 25 shows a diagram of the various token packets.

Figure 25. USB Token Packet Types



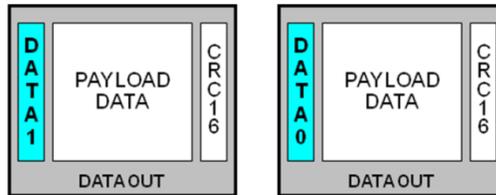
The SOF gives a way for devices to identify the beginning of a frame and synchronize with the host. They are also used to prevent a device from entering suspend mode (which it must do if 3 ms pass without an SOF). SOF packets are only seen on full- and high-speed devices and are sent every millisecond as seen in Figure 26. The SOF packet contains an 8-bit SOF PID, 11-bit frame count value (which rolls over when it reaches maximum value), and a 5-bit CRC. The CRC is the only error check used. A handshake packet does not occur for an SOF packet. High-speed communication goes a step further with microframes. With a High-Speed device, an SOF is sent out every 125 μ s and frame count is only incremented every 1 ms.

Figure 26. USB SOF in Full-Speed Device



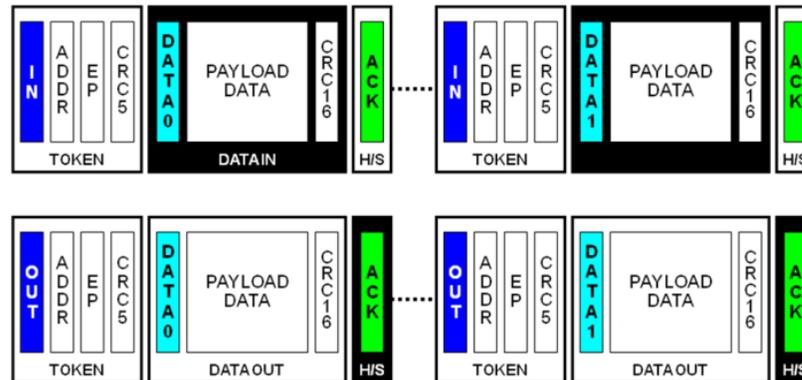
Data Packets: Data packets follow IN, OUT, and SETUP token packets. The size of the payload data ranges from 0 to 1024 bytes depending on the transfer type. The packet ID toggles between DATA0 and DATA1 for each successful data packet transfer, and the packet closes with a 16-bit CRC. The composition of a data packet can be seen in Figure 27.

Figure 27. USB Data Packets



The data toggle is updated at the host and the device for each successful data packet transfer. One advantage to the data toggle is that it acts as an additional error detection method. If a different packet ID is received than what is expected, the device will be able to know there was an error in the transfer and it can be handled appropriately. An example where the data toggle is used is if an ACK is sent but not received. In this instance, the sender updates the data toggle from '1' to '0' but the receiver does not. The receiver remains at '1'. This causes the host and device to be out of sync on the next data stage, which indicates an error. An example of the data toggle in a USB transfer can be seen in Figure 28. In this figure, and all other figures in this application note, white boxes represent the transaction is coming from the host and black boxes represent the transaction is coming from the device.

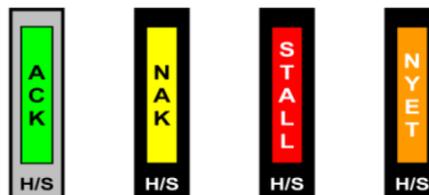
Figure 28. Data Toggle Example



Handshake Packets: Handshake packets conclude each transaction. Each handshake includes an 8-bit packet ID and is sent by the receiver of the transaction. Each USB speed has several options for a handshake response. Which ones are supported depend on the USB speed:

- **ACK:** Acknowledge successful completion. (LS/FS/HS)
- **NAK:** Negative acknowledgement. (LS/FS/HS)
- **STALL:** Error indication sent by a device. (LS/FS/HS)
- **NYET:** indicates the device is not ready to receive another data packet. (HS Only)

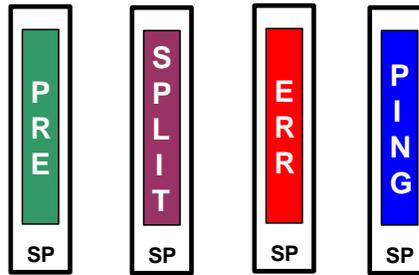
Figure 29. Handshake Packet Indicators



Special Packets: The USB specification defines four special packets.

- **PRE:** Is issued to hubs by the host to indicate that the next packet is low speed.
- **SPLIT:** Precedes a token packet to indicate a split transaction. (HS Only)
- **ERR:** Returned by a hub to report an error in a split transaction. (HS Only)
- **PING:** Checks the status for a Bulk OUT or Control Write after receiving a NYET handshake. (HS Only)

Figure 30. Special Packet Indicator



9.2 Transaction Types

USB transactions are how data from the host and the device get from point A to point B. There are a couple of different transaction types and they often use different names to represent the same concept. The three different transaction types are as follows.

9.2.1 IN/Read/Upstream Transactions

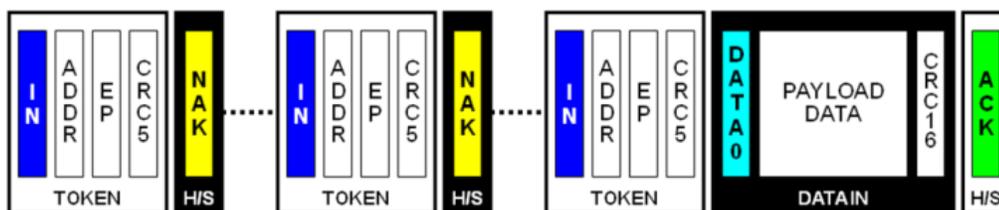
IN, Read, and Upstream are terms that refer to a transaction that is sent from the device to the host. These transactions are initiated by the host by sending an IN token packet. The targeted device responds by sending one or more data packets, and the host responds with a handshake packet. Figure 31 shows white boxes for transactions from the host and the black box for the transaction from the device.

Figure 31. IN/Read/Upstream Block Diagram



In Figure 32, the device responds with NAKs to show that it is not ready to send data when the host makes the request. The host continues to retry and the device responds with a data packet when it is ready. The host then acknowledges the receipt of the data with an ACK handshake.

Figure 32. IN Transaction Example



9.2.2 OUT/Write/Downstream Transactions

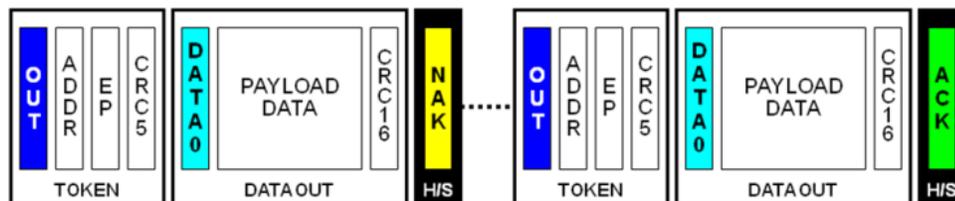
OUT, Write, and Downstream are terms that refer to a transaction that occurs from the host to the device. In this type of transaction, the host sends the appropriate token packet (either an OUT or SETUP), and follows with one or more data packets. The receiving device ends the transaction by sending the appropriate handshake packet. Figure 33 shows white boxes for transactions from the host and the black box for the transaction from the device.

Figure 33. OUT/Write/Downstream Block Diagram



In Figure 34, the host sends the OUT token packet and a DATA0 packet but receives a NAK from the device. The host then retries to send the data. Notice that the data toggle bit has not changed since the handshake was NAKed. With the next attempt to send data, the device responds with an ACK to indicate that the OUT transaction was successful.

Figure 34. OUT Transaction Example



9.2.3 Control Transactions

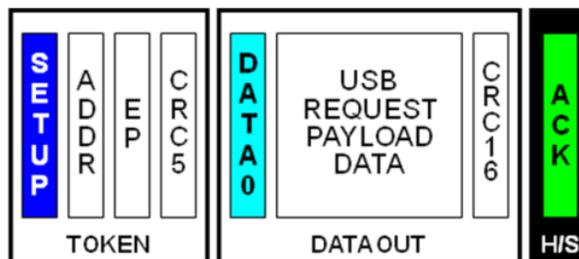
Control transfers identify, configure, and control devices. They enable the host to read information about a device, set the device address, establish configuration, and issue certain commands. A control transfer is always directed to the control endpoint of a device. Control transfers have three stages: the setup stage, (optional) data stage, and status stage. Figure 35 shows three stages are transferred by the host. The dotted line around the data stage shows that it is an optional transaction.

Figure 35. Control Transfer Block Diagram



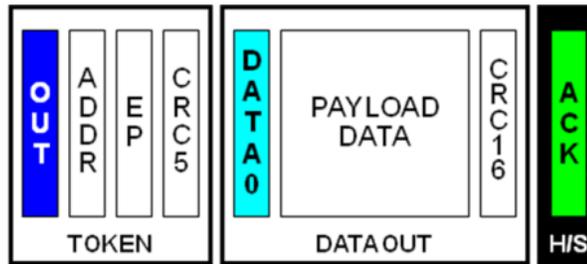
The setup stage (or setup packet) is only used in a control transaction. This packet sends USB requests from the host to the device and requires the data packet to contain an 8-byte USB request. The device must always acknowledge the setup stage, you cannot NAK a setup stage.

Figure 36. Setup Stage Transaction



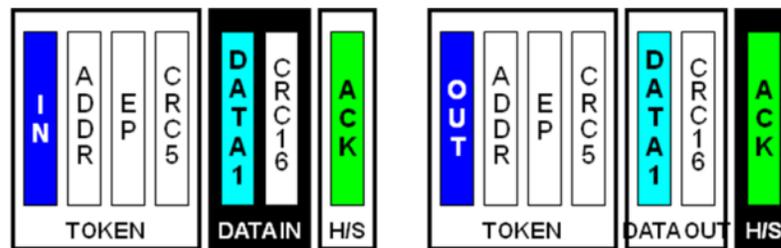
The data stage is optional in a control transaction. This stage can consist of multiple data transactions and is only required when a data payload is to be transferred between the host and device. Frequently, relevant data for the control transfer can be transferred in the setup stage.

Figure 37. Setup Stage Transaction



The final stage, the status stage, includes a single IN or OUT transaction that reports on the success or failure of the previous stages. The data packet is always DATA1 (unlike normal IN and OUT transactions that toggle between DATA0 and DATA1) and contains a zero length data packet. The status stage ends with a handshake transaction that is sent by the receiver of the preceding packet.

Figure 38. Status Stage Transaction



USB communications have three types of control transfers: control write, control read, and control no data. [Figure 39](#), [Figure 40](#), and [Figure 41](#) show examples of these transactions.

Figure 39. Control No Data Transaction

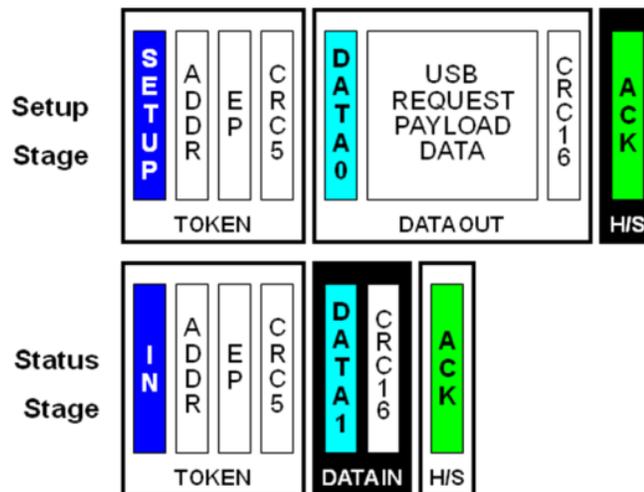


Figure 40. Example of Control Write Transaction

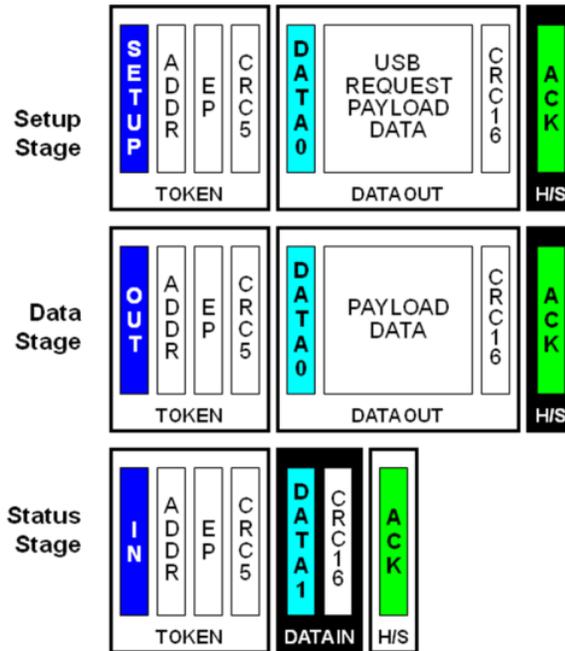
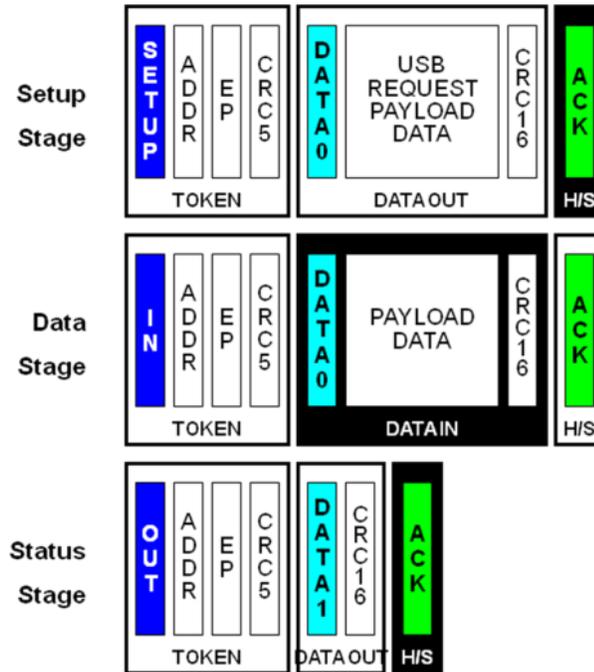


Figure 41. Example of Control Read Transaction



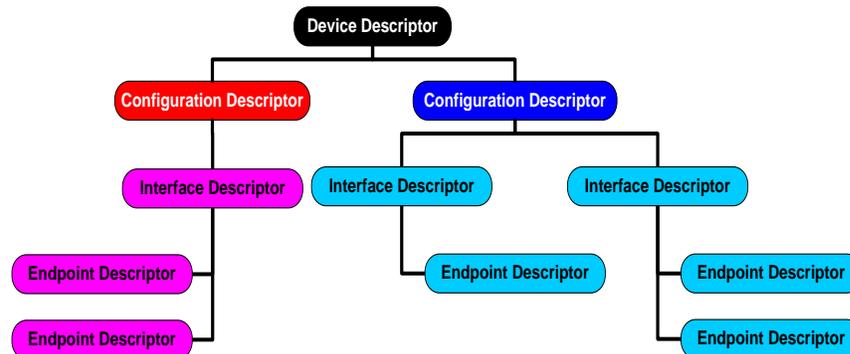
For more information on USB transfer types and their implementations in PSoC 3 and PSoC 5 devices, see the [AN56377 – PSoC® 3 / PSoC 5LP – Introduction to Implementing USB Data Transfer](#).

10 USB Descriptors

As described earlier, when a device is connected to a USB host, the device gives information to the host about its capabilities and power requirements. The device typically gives this information through a descriptor table that is part of its firmware. A descriptor table is a structured sequence of values that describe the device; these values are defined by the developer. All descriptor tables have a standard set of information that describes the device attributes and power requirements. If a design conforms to the requirement of a particular USB device class, additional descriptor information that the class must have is included in the device descriptor structure. [Appendix A](#) includes an example of a fully functional device descriptor for a PSoC USB device.

It is important to know that data fields are transmitted with the least significant bit first. Remember this when reading or creating your own descriptors. Many parameters are 2 bytes long. Make sure that the low byte is first followed by the high byte.

Figure 42. USB Descriptor Tree with Two Configurations



10.1 Device Descriptor

Device descriptors give the host information such as the USB specification to which the device conforms, the number of device configurations, and protocols supported by the device, Vendor Identification (also known as VID, which is something that each company gets uniquely from the USB Implementers Forum), Product Identification (also known as PID, different from a packet ID), and a serial number if the device has one. The device descriptor is where some of the most crucial information about the USB device is contained. [Table 5](#) shows the structure for a device descriptor.

Table 5. Device Descriptor Table

| Offset | Field | Size (Bytes) | Description |
|--------|--------------------|--------------|---------------------------------------------------|
| 0 | bLength | 1 | Length of this descriptor = 18 bytes |
| 1 | bDescriptorType | 1 | Descriptor type = DEVICE (01h) |
| 2 | bcdUSB | 2 | USB specification version (BCD) |
| 4 | bDeviceClass | 1 | Device class |
| 5 | bDeviceSubClass | 1 | Device subclass |
| 6 | bDeviceProtocol | 1 | Device Protocol |
| 7 | bMaxPacketSize0 | 1 | Max Packet size for endpoint 0 |
| 8 | idVendor | 2 | Vendor ID (or VID, assigned by USB-IF) |
| 10 | idProduct | 2 | Product ID (or PID, assigned by the manufacturer) |
| 12 | bcdDevice | 2 | Device release number (BCD) |
| 14 | iManufacturer | 1 | Index of manufacturer string |
| 15 | iProduct | 1 | Index of product string |
| 16 | iSerialNumber | 1 | Index of serial number string |
| 17 | bNumConfigurations | 1 | Number of configurations supported |

bLength is the total length in bytes of the device descriptor.

bcdUSB reports the USB revision that the device supports, which should be latest supported revision. This is a binary-coded decimal value that uses a format of 0xAABC, where A is the major version number, B is the minor version number, and C is the sub-minor version number. For example, a USB 2.0 device would have a value of 0x0200 and USB 1.1 would have a value of 0x0110. This is normally used by the host in determining which driver to load.

bDeviceClass, **bDeviceSubClass**, and **bDeviceProtocol** are used by the operating system to identify a driver for a USB device during the enumeration process. Filling in this field in the device descriptor prevents different interfaces from functioning independently, such as a composite device. Most USB devices define their classes in the interface descriptor, and leave these fields as 00h.

bMaxPacketSize reports the maximum number of packets supported by Endpoint 0. Depending on the device, the possible sizes are 8 bytes, 16 bytes, 32 bytes, and 64 bytes.

iManufacturer, **iProduct**, and **iSerialNumber** are indexes to string descriptors. String descriptors give details about the manufacturer, product, and serial number. If string descriptors exist, these variables should point to their index location. If no string exists, then the respective field should be assigned a value of zero.

bNumConfigurations defines the total number of configurations the device can support. Multiple configurations allow the device to be configured differently depending on certain conditions such as being bus powered or self powered. More details regarding this are discussed later.

10.2 Configuration Descriptor

This descriptor gives information about a specific device configuration such as the number of interfaces, if the device is bus-powered or self-powered, if the device can start a remote wake-up, and how much power the device needs. [Table 6](#) shows the structure for a configuration descriptor.

Table 6. Configuration Descriptor Table

| Offset | Field | Size (Bytes) | Description |
|--------|---------------------|--------------|----------------------------------------------------------------------------|
| 0 | bLength | 1 | Length of this descriptor = 9 bytes |
| 1 | bDescriptorType | 1 | Descriptor type = CONFIGURATION (02h) |
| 2 | wTotalLength | 2 | Total length including interface and endpoint descriptors |
| 4 | bNumInterfaces | 1 | Number of interfaces in this configuration |
| 5 | bConfigurationValue | 1 | Configuration value used by SET_CONFIGURATION to select this configuration |
| 6 | iConfiguration | 1 | Index of string that describes this configuration |
| 7 | bmAttributes | 1 | Bit 7: Reserved (set to 1) Bit 6: Self-powered Bit 5: Remote wakeup |
| 8 | bMaxPower | 1 | Maximum power required for this configuration (in 2 mA units) |

wTotalLength is the length of the entire hierarchy of this configuration. This value reports the total number of bytes of the configuration, interface, and endpoint descriptors for one configuration.

bNumInterfaces defines the total number of possible interfaces in this particular configuration. This field has a minimum value of 1.

bConfigurationValue defines a value to use as an argument to the SET_CONFIGURATION request to select this configuration.

bmAttributes defines parameters for the USB device. If the device is bus powered, bit 6 is set to 0, if the device is self powered, than bit 6 is set to 1. If the USB device supports remote wakeup, bit 5 is set to 1. If remote wakeup is not supported, bit 5 is set to 0.

bMaxPower defines the maximum power consumption drawn from the bus when the device is fully operational, expressed in 2 mA units. If a self-powered device becomes detached from its external power source, it may not draw more than the value indicated in this field.

10.3 Interface Association Descriptor (IAD)

This descriptor describes two or more interfaces that are associated with a single device function. The interface association descriptor (IAD) informs the host that the interfaces are linked together. For example, a USB UART has two interfaces associated with it: a control interface and a data interface. The IAD tells the host that these two interfaces are part of the same function, which is a USBUART, and falls under the communication device class (CDC). This descriptor is not required in all cases. Figure 43 shows how a single interface relates to a single device function. Figure 43 shows how two separate interfaces are linked to a particular device function. This is where the IAD is required. Table 7 shows the structure for an interface association descriptor.

Figure 43. Multiple Interfaces with Multiple Functions

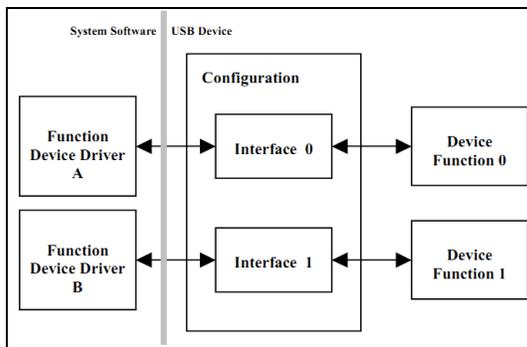


Figure 44. Multiple Interfaces with Single Function

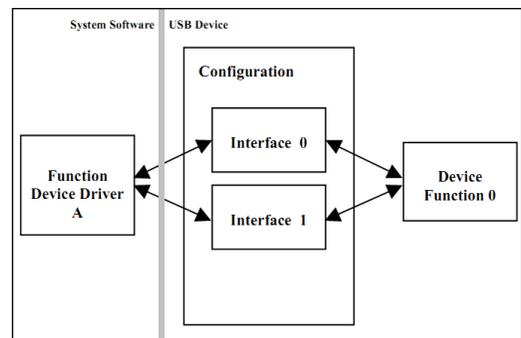


Table 7. Interface Association Descriptor Table

| Offset | Field | Size (Bytes) | Description |
|--------|-------------------|--------------|---------------------------------------------------------------------|
| 0 | bLength | 1 | Descriptor size in bytes |
| 1 | bDescriptorType | 1 | Descriptor type = INTERFACE ASSOCIATION (0Bh) |
| 2 | bFirstInterface | 1 | Number identifying the first interface associated with the function |
| 3 | bInterfaceCount | 1 | The number of contiguous interfaces associated with the function |
| 4 | bFunctionClass | 1 | Class code |
| 5 | bFunctionSubClass | 1 | Subclass code |
| 6 | bFunctionProtocol | 1 | Protocol code |
| 7 | iFunction | 1 | Index of string descriptor for the function |

10.4 Interface Descriptor

An interface descriptor describes a specific interface within a configuration. The number of endpoints for an interface is identified in this descriptor. The interface descriptor is also where the USB Class of the device is declared. There are many predefined classes that a USB device can be, many of which are listed in Table 12. A USB device class identifies the device functionality and aids in the loading of a proper driver for that specific functionality. Table 8 shows the structure for an interface descriptor.

Table 8. Interface Descriptor Table

| Offset | Field | Size (Bytes) | Description |
|--------|--------------------|--------------|----------------------------------------------------------------|
| 0 | bLength | 1 | Length of this descriptor = 9 bytes |
| 1 | bDescriptorType | 1 | Descriptor type = INTERFACE (04h) |
| 2 | bInterfaceNumber | 1 | Zero based index of this interface |
| 3 | bAlternateSetting | 1 | Alternate setting value |
| 4 | bNumEndpoints | 1 | Number of endpoints used by this interface (not including EP0) |
| 5 | bInterfaceClass | 1 | Interface class |
| 6 | bInterfaceSubclass | 1 | Interface subclass |
| 7 | bInterfaceProtocol | 1 | Interface protocol |
| 8 | iInterface | 1 | Index to string describing this interface |

10.5 Endpoint Descriptor

Each endpoint used in a device has its own descriptor. This descriptor gives the endpoint information that the host must have. This information includes direction of the endpoint, transfer type, and maximum packet size. [Table 9](#) shows the structure for an endpoint descriptor.

Table 9. 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 isochronous endpoints, ignored for control or bulk) |

10.6 String Descriptor

The string descriptor is another optional descriptor and gives user readable information about the device. Possible information contained in the descriptor is the name of the device, the manufacturer, the serial number, or names for the various interfaces or configurations. If strings are not used in a device, any string index field of the descriptors mentioned earlier must be set to 00h.

Strings are defined using UNICODE UTF16LE encodings and can support multiple languages with a language ID code. In a Windows system, these strings can be seen in the device manager. [Table 10](#) shows the structure for a string descriptor.

Table 10. String Descriptor Table

| Offset | Field | Size (Bytes) | Description |
|--------|----------------------------|--------------|----------------------------------------------------|
| 0 | bLength | 1 | Length of this descriptor = 7 bytes |
| 1 | bDescriptorType | 1 | Descriptor type = STRING (03h) |
| 2..n | bString -or- wLangID | Varies | Unicode encoded text string -or- LANGID code |

The string descriptor has two possible forms. The first string descriptor contains a value for the language ID, **wLangID**, which contains one or more two byte ID codes that indicate the languages in which the strings are. USB-IF gives a document that defines many different ID codes. U.S. English for example is 0409h. For more ID codes, see the [USB-IF LANGID](#) page. All string descriptors that occur after wLangID use **bString**, which is a string field that contains a Unicode string (UTF16LE) and uses 2 bytes to represent each character.

10.7 Other Miscellaneous Descriptor Types

Report Descriptors: A USB device class may require an extended set of descriptor information. Developers must make certain that any additional descriptor information required by a USB device class is included in the descriptor file. For example, with the HID class the developer must include report descriptors that further define the device attributes. If additional descriptors are required, the descriptor format is present in the class definition specification or other class supporting documentation. For additional information on Report Descriptors, see [AN57473 - USB HID Basics with PSoC[®] 3 and PSoC 5LP](#) and [AN58726 - USB HID Intermediate with PSoC[®] 3 and PSoC 5LP](#).

MS OS Descriptor: Microsoft has a descriptor called the Microsoft OS Feature Descriptor (also called the MS OS descriptor) that is used in vendor-specific devices. This descriptor gives Microsoft Windows specific information such as special icons, registry settings, help files, and URLs. The MS OS descriptor contains a string and one or more feature descriptors. During enumeration, Windows requests the string descriptor, which contains an index of EEh and an embedded signature. If the device supports the MS OS descriptor, Windows requests additional information after receiving the string descriptor. If the device does not support the MSOS descriptor, the device returns a STALL as the handshake. More information on MS OS descriptors is at the [MSDN Microsoft OS descriptors](#) landing page.

Device_Qualifier Descriptor: Another descriptor seen in USB is a Device_Qualifier Descriptor. This describes information about a High-Speed capable device that changes if the device operates at another speed and is required by devices that support both speed configurations. If a device operates at Full-Speed when this descriptor is requested, it tells the host how the device would operate differently if it were operating at High-Speed. The same is true if the descriptor is requested while the device is operating at High-Speed. The descriptor read tells the host about a Full-Speed configuration. If this descriptor is requested and the device supports only Full-Speed, the proper action is to respond with a STALL. Otherwise, the descriptor information will be provided to the host upon request. For more information on this descriptor, see Section 9.6.2 of the USB specification.

BOS Descriptor: Another descriptor seen in USB 2.0 devices that support Link Power Management (LPM) is the Binary device Object Store (BOS) descriptor. In the PSoC family of devices that have USB 2.0 device support, only the PSoC 4200L family of devices support the LPM feature, and consequently the BOS descriptor. LPM is an improvement over the USB suspend mode that enables devices to enter and exit low-power modes with transition latencies of the order of tens of microseconds compared to the 3-20 ms latency associated with suspend mode entry/exit.

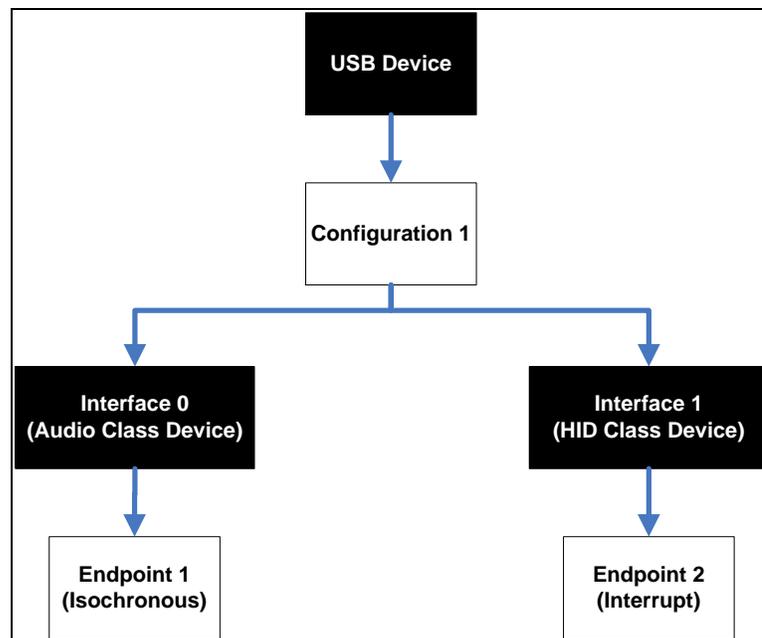
Any USB 2.0 device that supports the LPM feature has to report its LPM capabilities through the USB 2.0 extension descriptor. The USB 2.0 extension descriptor is part of the BOS descriptor. The value of the bcdUSB field in the standard USB 2.0 device descriptor is used to indicate that the device supports the request to read the BOS descriptor through the GetDescriptor(BOS) request. Devices that support the BOS descriptor must have bcdUSB value of 0201H or larger. See Section 9.6.2 of the USB 3.1 specification for details on the BOS descriptor and USB 2.0 extension descriptor.

10.8 Using Multiple USB Descriptors

USB devices have only one device descriptor. However, a device can have multiple configurations, interfaces, endpoints, and string descriptors. When a device is enumerated, one of the final stages is to read the device descriptors and make a decision on which device configuration to enable. Only one configuration can be enabled at a time. An example of this is a design that has one configuration that is used when the device is self-powered and another configuration when the device is bus-powered. The overall USB functionality may be different for the self-powered device than for the bus-powered device. Having multiple configurations and multiple configuration descriptors allows the option to implement this ability.

At the same time, a device can have multiple interfaces, thus multiple interface descriptors. A USB device with multiple interfaces that perform different functions is called a composite device. An example is a USB audio head set. In the head set, you have a single USB device with two interfaces. One interface is for the audio side of the headset and another interface may be the controls to adjust the volume. Multiple interfaces can be active at the same time. [Figure 45](#) shows a diagram of the ability to split two interfaces under a single USB device.

Figure 45. Multiple Interface Settings Diagram



Finally, each interface can have multiple configurations. These multiple configurations are called alternate settings. One possible application is to allow the ability to alter endpoint configuration on a device to reserve different amounts of bandwidth. For example, in one alternate setting, a device can have its endpoints configured to bulk, which has no guaranteed bus bandwidth; in another alternate setting, it can have the endpoints configured for Isochronous, which has guaranteed bus bandwidth. This concept is shown in [Figure 46](#).

Figure 46. Multiple Interface Settings Diagram

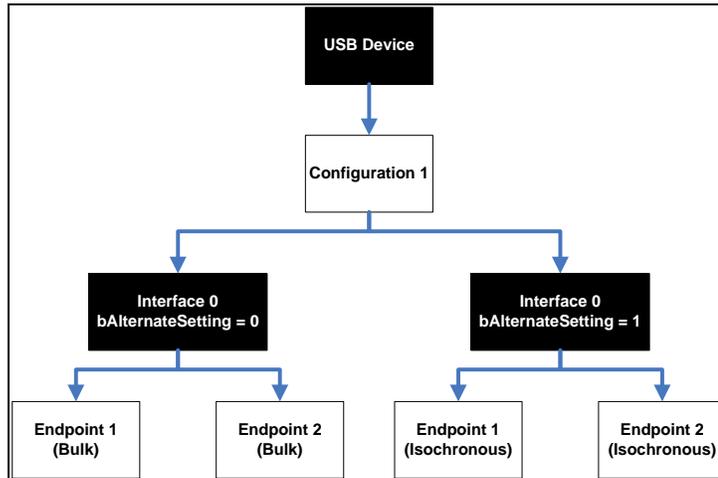
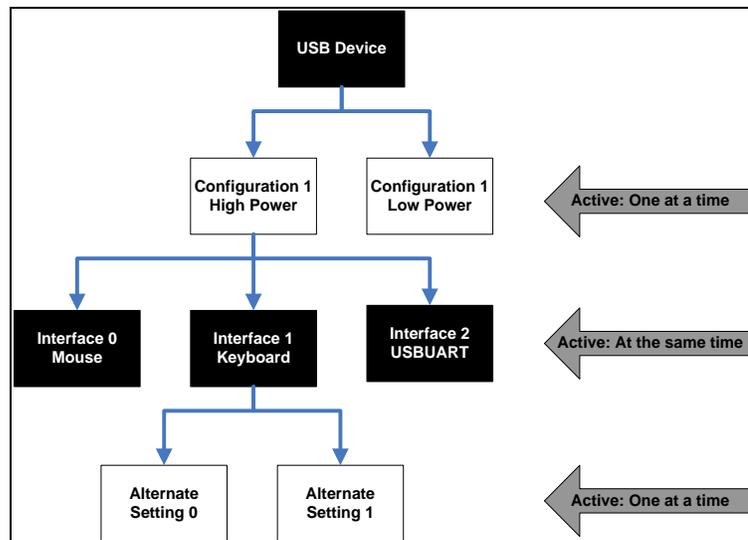


Figure 47 shows an overall diagram view of the customization options that can be used to create a highly customizable USB device to suit a variety of configurations options.

Figure 47. Configuration Diagram



11 USB Class Devices

The USB Implementers Forum has a list of recognized and approved [USB device classes](#). The most common device classes are

- Human Interface Device (HID)
- Mass Storage Device (MSD)
- Communication Device Class (CDC)
- Vendor (Vendor Specific)

There are several considerations to think about when developing an application for a certain class. First, each class has a fixed maximum bandwidth. Second, each class has limitations on the supported transfer types and certain commands that must be supported. However, the biggest advantage to using a predefined USB device class is the cross platform support across various operating systems. All major operating systems include a driver in the OS for most of the predefined USB classes that eliminates the need to create a custom driver. Table 11 shows some of the more common drivers that are used with Cypress products and some of the capabilities of those drivers.

Table 11. USB Device Class Driver Features

| Feature | HID | CDC | WinUSB | LibUSB | CYUSB |
|-----------------------------------|----------|-----------|--------------------------|---------|---------|
| Driver Support in Windows | Yes | Need .inf | Need .inf ^[1] | No | No |
| Support for 64-Bit | Yes | Yes | Yes | Yes | Yes |
| Support for Control Transfers | Yes | No | Yes | Yes | Yes |
| Support for Interrupt Transfer | Yes | No | Yes | Yes | Yes |
| Support for Bulk Transfers | No | Yes | Yes | Yes | Yes |
| Support for Isochronous Transfers | No | Yes | No | Yes | Yes |
| Maximum Speed (Full-Speed) | ~64 KB/s | ~80 KB/s | ~1 MB/s | ~1 MB/s | ~1 MB/s |

[1].WinUSB is not native to Windows XP; it must be installed with the WinUSB co-installer.

Devices that do not meet the definition of a specific USB device class are called vendor-specific devices. These devices allow developers to create applications with their own creativity and customization options, which are not bound by a specific USB class, but still conform to the USB specification. Devices that fall under a vendor-specific device use WinUSB, CYUSB, LibUSB, or another type of vendor-specific driver. The advantage to using WinUSB is that it is Windows own vendor-specific driver and does not need to undergo Windows Hardware Quality Labs (WHQL) testing for driver signing. WHQL testing is discussed later in this application note. LibUSB is an open source driver project with support for Windows, Mac, and Linux operating systems. CyUSB is Cypress' own vendor specific driver. The advantage to using this driver in an application is the broad range of example applications, supporting documentation, and direct support from Cypress.

In the [USB Descriptors](#) section, notice that the fourth byte in the device descriptor and the sixth byte in the interface descriptor are where the class of the USB device is defined. The USB specification defines many different USB classes and the device class codes that go along with them. Table 12 shows some USB class codes that can be used in these bytes to give an idea of the various USB classes that are available.

Table 12. USB Class Codes

| Class | Usage | Description | Examples |
|-------|-----------|---------------------------------|-----------------------------------------------------------------------------------------|
| 00h | Device | Unspecified | Device class is unspecified, interface descriptors are used to determine needed drivers |
| 01h | Interface | Audio | Speaker, microphone, sound card, MIDI |
| 02h | Both | Communications and CDC Control | Modem, ethernet adapter, Wi-Fi adapter |
| 03h | Interface | Human Interface Device (HID) | Keyboard, mouse, joystick |
| 05h | Interface | Physical Interface Device (PID) | Force feedback joystick |
| 06h | Interface | Image | Camera, scanner |
| 07h | Interface | Printer | Printers, CNC machine |
| 08h | Interface | Mass Storage | External hard drives, flash drives, memory cards |
| 09h | Device | USB Hub | USB hubs |
| 0Ah | Interface | CDC-Data | Used in conjunction with class 02h. |
| 0Bh | Interface | Smart Card | USB smart card reader |
| 0Dh | Interface | Content Security | Fingerprint reader |

| Class | Usage | Description | Examples |
|-------|-----------|----------------------|----------------------------------------------------------------------------------|
| 0Eh | Interface | Video | Webcam |
| 0Fh | Interface | Personal Healthcare | Heart rate monitor, glucose meter |
| DCh | Both | Diagnostic Device | USB compliance testing device |
| E0h | Interface | Wireless Controller | Bluetooth adapter |
| EFh | Both | Miscellaneous | ActiveSync device |
| FEh | Interface | Application Specific | IrDA Bridge, Test & Measurement Class (USBTMC), USB DFU (direct firmware update) |
| FFh | Both | Vendor Specific | Indicates a device needs vendor specific drivers |

12 USB Enumeration and Configuration

Normally, developers look at enumeration as a single process in a USB device. Enumeration is actually one part in a three-stage process: dynamic detection, enumeration, and configuration. Dynamic detection is the recognition of a change in the state of a USB port. In [Figure 17](#) you see that there are pull-down resistors on the host/hub side. When a device is attached, one of these lines is pulled high depending on device speed. It is with this voltage transition that the host/hub detects the change of the bus port. Enumeration directly follows the device detection, and is the process of assigning a unique address to a newly attached device. Configuration is the process of determining a device's capabilities by an exchange of device requests. The requests that the host uses to learn about a device are called standard requests and must support these requests on all USB devices.

The entire enumeration process is described in the following sections.

12.1 Dynamic Detection

Step 1: The device is connected to a USB port and detected. At this point, the device can draw up to 100 mA from the bus. The device is currently in the powered state.

Step 2: The hub detects the device by monitoring voltages on the ports. A hub has pull-down resistors on the D+ and D- lines as seen in [Figure 17](#). As mentioned earlier, there is a pull-up resistor on either the D+ or D- line depending on device speed. By monitoring the voltage transition on these lines, the hub detects if a device is attached.

12.2 Enumeration

Step 3: The host learns of the newly attached device by using an interrupt endpoint to get a report about the hub's status. This includes changes in port status. After the hub tells the host about the device detection, the host issues a request to the hub to learn more details about the status change that occurred using the GET_PORT_STATUS request.

Step 4: After the host gathers this information, it detects the speed of the device using the method mentioned in the USB Speeds. Initially, only Full-Speed or Low-Speed is detected by the hub by detecting if the pull-up resistor is on the D+ or D- line. This information is then reported to the host by another GET_PORT_STATUS request.

Step 5: The host issues a SET_PORT_FEATURE request to the hub asking it to reset the newly attached device. The device is put into a reset state by pulling both the D+ and D- lines down to GND (0 V). Holding these lines low for greater than 2.5 us issues the reset condition. This reset state is held for 10 ms by the hub.

Step 6: During this reset, a series of J-State and K-State occurs to determine if the device supports High-Speed. During this reset state, if the device supports High-Speed, it issues a single K-State. A High-Speed hub detects this K-State and responds with a sequence of J and K states to form a "KJKJKJ" pattern. The device detects this pattern and removes its pull-up resistor from its D+ line. This step is skipped on Low-Speed and Full-Speed devices.

Step 7: The host then checks to see if the device is still in a reset state by issuing a GET_PORT_STATUS request. If the request reports that the device is still in reset, then the host continues to issue the request until it receives word that the device is out of reset. After the device leaves reset, it is in the default state as mentioned in the [USB Power](#) section. The device can now respond to requests from the host in the form of control transfers to its default address of 00h. All USB devices start with this default address. Only one USB device can have this address at a time; this is why when you connect multiple USB devices to a port at the same time, they enumerate sequentially and not simultaneously.

Step 8: The host begins the process of learning more information about the device. It starts by learning the maximum packet size of the default pipe (Endpoint 0). The host starts by issuing a GET_DESCRIPTOR request to the device. The device begins to send the descriptors discussed in the [USB Descriptors](#) section of the application note. In the device descriptor, the eighth byte (bMaxPacketSize0) contains information about the maximum packet size for EP0. A Windows host requests 64-bytes, but after only receiving 8 bytes of the device descriptor, it moves onto the status stage of the control transfer and requests that the hub reset the device. The USB specification requires that a device return at least 8 bytes of the device descriptor, when requested, if the device has the default address of 00h. The reason for requesting the 64-bytes is to avoid unpredictable behavior from the device. Additionally, the reason for performing a reset after only receiving 8 bytes is an artifact of early USB devices. In the early life of USB, some devices did not respond properly when a second request for the device descriptor occurred. To solve this problem, a reset after the first device descriptor request was required. Regardless, the 8 bytes that were transferred was enough to get the require information about the bMaxPacketSize0.

Step 9: The host applies an address to the device with the SET_ADDRESS request. The device completes the status stage of this request using the default 00h address before using the newly assigned address. All communication beyond this point will use the new address. The address may change if the device is detached from a port, the port is reset, or the PC reboots. The device is now in the address state.

12.3 Configuration

Step 10: After the device returns from its reset, the host issues a command, GET_DESCRIPTOR, using the newly assigned address, to read the descriptors from the device. However, this time all the descriptors are read. The host uses this information to learn about the device and its abilities. This information includes the number of peripheral interfaces, power connection method, and the required maximum power. The host starts by requesting the device descriptor, but this time it receives the entire descriptor and not just a partial version. Next the host issues another GET_DESCRIPTOR command asking for the configuration descriptor. This request not only returns the configuration descriptor, but all other descriptors associated with it such as the interface descriptor and the endpoint descriptor. A Windows PC first asks for just the configuration descriptor (9 bytes), than it issued a second GET_DESCRIPTOR request for the configuration descriptor and all other associated descriptors with that configuration (interface and endpoint descriptors).

Step 11: For the host PC to successfully use the device, a Windows PC in this case, the host must load a device driver. The host searches for a driver to manage communication between itself and the device. Windows uses its .inf files to locate a match for the devices Product ID and Vendor ID. Device release version numbers can optionally be used. If Windows cannot find a match, then it looks at the driver from a different perspective by looking for a match with any class, subclass, and protocol retrieved from the device. If a device was previously enumerated, Windows uses its registry to search for the proper driver. When a driver is identified, the host may request additional descriptors that are specific to the device class or request that descriptors are resent.

Step 12: After all descriptors are received, the host sets a specific device configuration using the SET_CONFIGURATION request. Most devices have only one configuration. Devices that support multiple configurations can allow the user or the driver to select the proper configuration.

Step 13: The device is now in the configured state. It took on its proprieties that were defined with the descriptors. The defined maximum power can be drawn from V_{BUS} and the device is now ready for use in an application.

13 Debugging USB Designs

Picture this situation: you have spent hours, days, or weeks developing a design that uses USB and now the time has come to test the design. The firmware is loaded and you go to plug the device into the host. As you stare at the prototype, you realize that nothing is happening; something went wrong. You now need to do some debug to check what happened. The question racing through your mind is where to start. There are several tips and tools to use when attempting to debug a design and this section is intended to help you through the debug process.

13.1 Debugging on the Host Side

When a USB device does not seem to be working properly, the first step is to start the debug process from the host side, because the host is responsible for initiating the communication with the device. Based on what is observed, a progression to debugging the device may be required. If the host being used is running Windows, a list of attached devices and their status can be found in the Device Manager (shown in [Figure 48](#)) by following these steps:

If the host being used is running **Windows XP/Vista**, a list of attached devices can be found by following these steps:

Step 1: Click **Start > System > Hardware > Device Manager**.

If the host being used is running **Windows 7**, a list of attached devices can be found by following these steps:

Step 1: Click **Start > Control Panel**.

Step 2: Select **System > Device Manager**. Note that you may need to change the “View by:” option to “Large Icons” or “Small Icons”. The options provided will be different for “Category”.

If the host being used is running **Windows 8/8.1**, a list of attached devices can be found by following these steps:

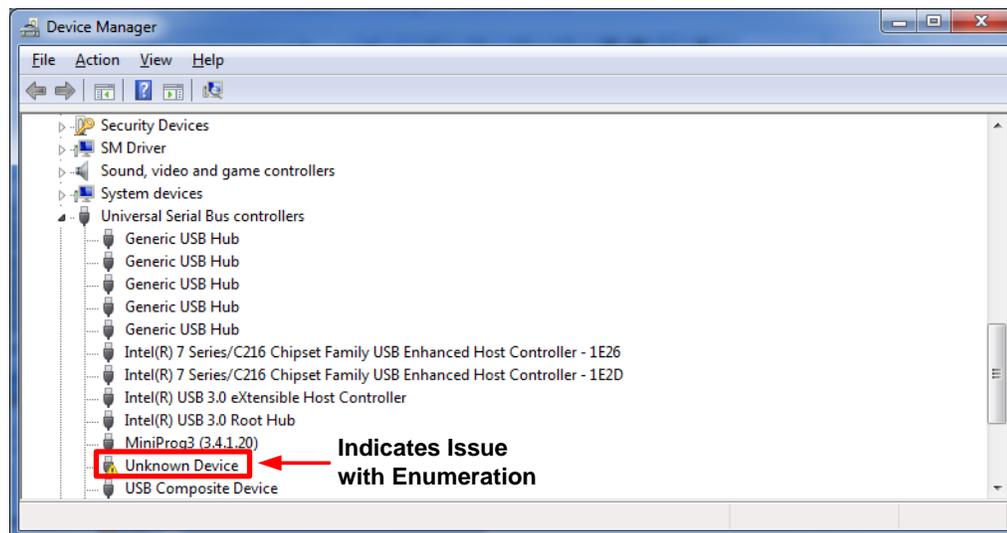
Step 1: Click the Desktop Tile to open Desktop.

Step 2: Click “>>” at the bottom right Taskbar.

Step 3: Select **Control Panel** from the menu that appears.

Step 4: Select **Device Manager** from the list of control panel items.

Figure 48. Windows Device Manager



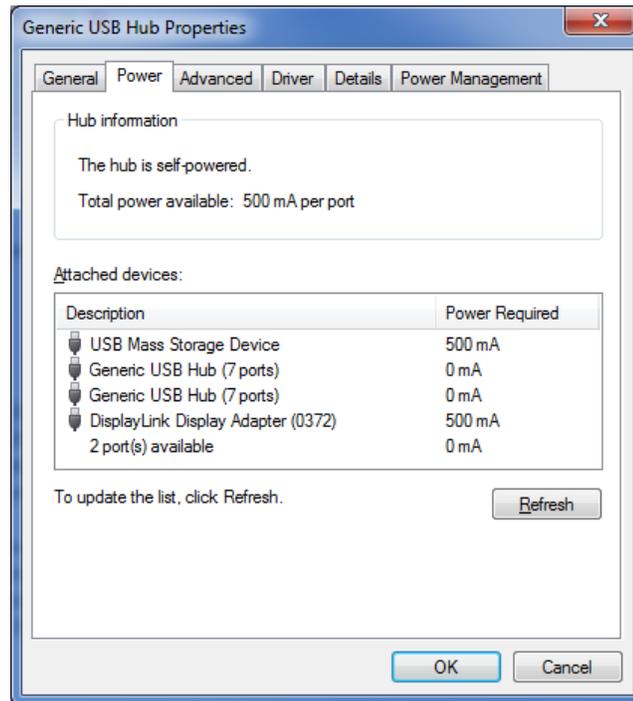
When there is an issue on the host side with a USB device, it is most commonly related to one of the two things: malfunctioning or improperly configured hardware or malfunctioning or missing device drivers. Using Device Manager, you can tell several things about the device connected.

Unknown Device: Similar to what is seen in [Figure 48](#), this issue can signify several different things. First, it can indicate that the device does not have a properly installed driver. Often, it will not list the “Unknown Device” under the “Universal Serial Bus controllers” branch, but rather under a “Other devices” branch. When this occurs, double clicking on the device, which opens the properties of the device, will show an Error Code 1 or Error Code 10. Many error codes can be reported by the device manager. Microsoft provides a list, description, and possible solution for the error codes to help when needed. The link can be found [here](#).

Additionally, a driver that was not installed or configured correctly can cause issues. This sometimes happens when a USB device is removed while Windows is trying to install and configure the hardware. Right-click on the device, select “Uninstall”, and unplug the device once complete. Plug the device back in. You may even need to force the driver update by right-clicking and selecting “Update Driver Software”.

Device is not listed: In the instance that you cannot find the device at all in Device Manager, the first thing to make sure is that the device is not exceeding the hub power requirements. Under the “Universal Serial Bus controllers” tree, you will see one or more “USB Root Hub” or “Generic USB Hub”. Double-click on them and select the “Power” tab. In there, you will see the attached device and the amount of power they each require. An example can be seen in [Figure 49](#). Additionally, the total power available will be listed. Ensure that the required power does not exceed the available power.

Figure 49. Device Manager - USB Power Properties



If that does not solve the problem, try a different USB port and/or cable. There is always the possibility that the contacts or some cabling has gone bad. Lastly, there is software provided by Cypress, such as the CY SuiteUSB/EZ-USB, that can be used to provide useful tools for debug without needing to develop test applications on the host side. Bundled with the software are the following tools:

Bulkloop: Simple tool that provides a loopback of data through a bulk transfer. Users can define a data pattern to send and the tool will inform the user how many bytes were transferred OUT and how many were transferred IN.

USB Data Streamer: Intended for isochronous data transfers. This tool is used to test the bandwidth and speed capabilities of a USB device by sending a constant stream of data. The user can configure the number of packets per transfer and the tool will keep track of the number of packets dropped, the number of packets received successfully, and throughput in KB/s.

USB Console: Intended to communicate with USB devices that are using the CyUSB.sys driver. Provides detailed information about the device using the driver and provides advanced control options to interface with the device.

USB Control Center: Intended to communicate with USB devices that are served by the Cypress CyUSB.sys driver, Windows Mass Storage driver (usbstor.sys), or Windows HID driver (hidusb.sys). This tool provides a capability to view attached USB devices, similar to Device Manager, while also providing a mechanism to communicate with various device endpoints by sending or requesting data.

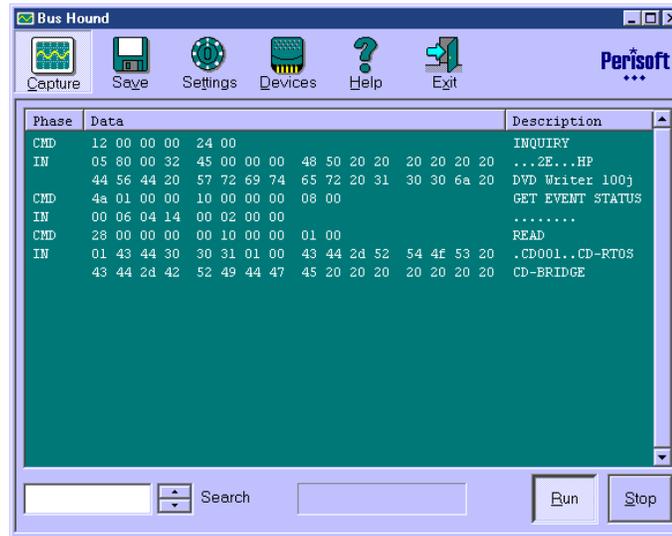
13.2 Debugging the Communication

When attempting to debug a USB design, there will come a time where one may need to view the USB traffic. To accomplish this, specialized tools are available. These tools come in two forms: hardware-based, where dedicated devices are attached to the system under test, and software-based, where the software resides on the host hardware. A USB analyzer allows you to record the bus traffic that is sent between the device and the host. They decode the USB traffic information such as enumeration, signaling errors, and general data transfer into an easy to read display. This provides an easy way to debug USB design issues.

Software analyzers can be used to capture device data transfers and protocols. They can also be used to send commands to the device. Software analyzers work by replacing the USB software stack on the host machine. This is a required step in order to monitor the USB traffic. As a result, the software analyzer will be dependent on the host hardware with regards to what information is available to be presented. Software analyzers have the advantage of being lower-priced than their hardware counter parts.

However, at a lower cost come some disadvantages. First, a host cannot monitor the USB traffic without being part of the bus itself. This means observing conditions such as suspend or reset is not possible. Additionally, depending on the host hardware and software stack, there are other bus conditions that the host-controller handles without software involvement. One final disadvantage is related to timing. The accuracy of timing with a software analyzer is dependent on the host operating system, which could induce varying levels of inaccuracies.

Figure 50. Bus Hound Software Analyzer by Perisoft



Hardware analyzers are more commonly used and are manufactured by a wide variety of companies. These devices are custom-manufactured pieces of hardware that are inserted between the host and the device to monitor the bus traffic. Some of these devices even have the capability to generate bus traffic if there is a specific condition that is trying to be produced. Prices for these types of devices can vary greatly depending on manufacturer, desired USB specification (USB 2.0 versus USB 3.0), and traffic generation capabilities. In general, these analyzers cost more than their software counterparts, but address the disadvantages that software analyzers face. While there are several companies that produce hardware analyzers, two more common companies are TotalPhase and LeCroy.

Figure 51. Examples of USB Analyzers



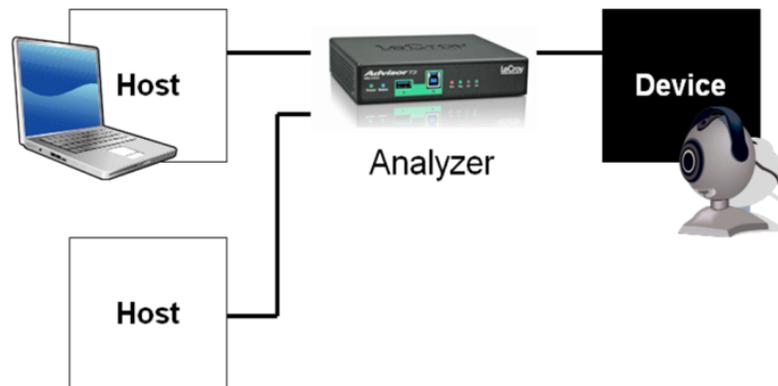
How these devices are used is quite simple and can be used in one of two ways. In one application, the USB analyzer is inserted between the host and the device being tested, as shown in Figure 52. The host will be responsible for running the application that is associated with the device under test while also running the analyzer software that will be used to view the traffic.

Figure 52. Using Analyzer with Single Host



In a second setup, two PCs can be used where one PC acts as the host for the device under test while the analyzer is connected to a second host responsible for monitoring the bus activity, as shown in [Figure 53](#). You may wonder why one analyzer configuration would be used over another. The answer is that there are situations where there is a need to monitor suspend and resume activities for the device being tested. Because the host has to be put into a suspend state, you will need a separate host to continue monitoring the traffic.

Figure 53. Using Analyzer with Dual Host



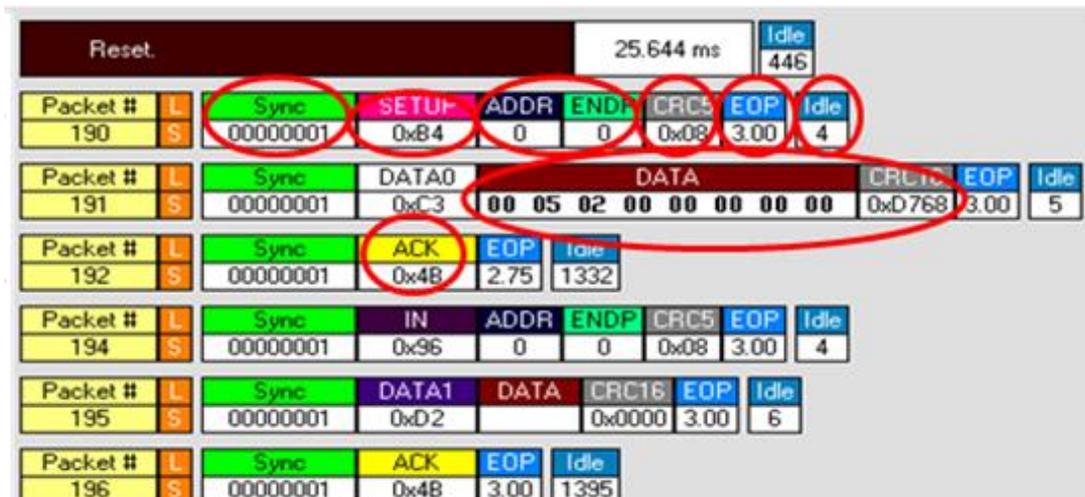
The prices of these analyzers can range from several hundred dollars to several thousand dollars. Note that if the application being designed does not require USB 3.0, money can be saved by purchasing a USB analyzer that only supports USB 2.0. Additionally, even more money can be saved with only purchasing a Low Speed-only or Full-Speed-only analyzer.

Hardware-based USB analyzers are not solely limited to a dedicated hardware, but can also be found on some higher-end oscilloscopes. Oscilloscopes such as these even have the capability to run the USB electrical tests performed during USB compliance testing.

Now that the various types of USB analyzers are understood, the next step is to look at what kind of information can be expected to be seen with an analyzer. Earlier in the [USB Enumeration and Configuration](#) section, the various steps of enumeration were discussed. Using a bus analyzer, the enumeration process can be observed. An example of a USB bus analyzer trace is shown in [Figure 54](#). [Appendix B](#) shows the enumeration of the USB device on a USB bus analyzer, whose descriptors are in [Appendix A](#). Using what you have just read in the [USB Enumeration and Configuration](#) section, and the information in [Appendix A](#), you have enough instructions to follow and understand the bus analyzer capture.

Be aware that the enumeration sequence shown in [Appendix B](#) is one of many possible enumeration event examples. This particular enumeration sequence uses the descriptors in [Appendix A](#) and enumeration was done on a Windows XP computer with the WinUSB driver. What you see varies depending on the particular device's descriptors, the operating system being enumerated on, and the drivers used.

Figure 54. USB Analyzer Output Example



| Packet # | L | S | Sync | Token | ADDR | ENDP | CRC5 | EOP | Idle | |
|----------|---|---|----------|-------|------|-------------------------|--------|------|------|---|
| 190 | | | 00000001 | SETUP | 0 | 0 | 0x08 | 3.00 | 4 | |
| 191 | | | 00000001 | DATA0 | 0xC3 | 00 05 02 00 00 00 00 00 | 0xD768 | 3.00 | 5 | |
| 192 | | | 00000001 | ACK | | | | 2.75 | 1332 | |
| 194 | | | 00000001 | IN | 0x96 | 0 | 0 | 0x08 | 3.00 | 4 |
| 195 | | | 00000001 | DATA1 | 0xD2 | | 0x0000 | 3.00 | 6 | |
| 196 | | | 00000001 | ACK | | | | 3.00 | 1395 | |

To help understand what Figure 54 is showing, let us focus specifically on Packet #190, Packet #191, and Packet #192.

■ Packet #190

- **L/S:** Implies Low-Speed
- **Sync:** Synchronizes the sending of packets between the host and the device.
- **SETUP:** Implies it is a SETUP token.
- **ADDR:** The current address is zero because it is the first transfer to the device. All control transfers are initiated from endpoint 0, address 0.
- **ENDP:** End point- gives the endpoint location. ENDP 0 implies it is a control transfer.
- **CRC5:** CRC 5 bits - detects errors in tokens.
- **EOP:** End of Packet - Signifies that the packet has ended.

■ Packet #191

- **DATA0:** Implies it is a data token
- **CRC 16:** 16 bit CRC check for data
- **Idle:** The time between the current packet and the previous packet.

■ Packet #192

- **ACK:** Indicates the successful completion of the data packet.

Bus analyzer traces are often used in other Cypress PSoC USB application notes. You must understand the information that is displayed on the bus analyzer trace to fully understand the information that is being conveyed with their help. These analyzers have the ability to show different levels of abstraction from a high level transaction view to a low level D+ and D- waveform view. It is up to you to determine how deeply you want to analyze data when using a bus analyzer.

That being said, a bus analyzer is an extremely valuable tool when debugging a USB once you know what to look for, since the analyzer can show information related to the packets being sent, the handshakes, and the timing. When USB basics are understood, the user will be able to look at what the analyzer captures, understand what is missing, and hone in on how to correct the issue.

13.3 Debugging on the Device Side

Capabilities of debugging on the device side will depend on the capabilities of the device and the development tools. For PSoC, PSoC Designer™ and PSoC Creator™ can be used to place breakpoints while inspecting memory locations and register configurations. The same stand true for the FX-series of USB parts along with Keil uVision and Eclipse. This provides a powerful method to make sure that ISRs are triggered and that commands are properly sent to the device.

Cypress provides Technical Reference Manuals for various products that will outline the various registers and the effect they have on the system. Those documents in conjunction with the debuggers in the IDEs will enable you to make sure that the device is working as expected.

14 Acquiring a VID and PID

To sell or market a USB product you must have a Vendor ID. This means that you must acquire one from USBIF. It is important to make sure that you get a Vendor ID for your company to avoid legal consequences. There are two options on how to acquire a VID:

- Become a member of USB-IF and pay the annual membership dues. At the time of this writing, the dues were \$4000/yr. Membership has several added benefits. Some of the most important include the eligibility to participate in free USB-IF sponsored quarterly Compliance Workshops, a Free Vendor ID (if one has not been previously assigned), a waived logo-administration fee when joining the USB-IF logo program, and the eligibility to participate in Device Working Groups.
- Purchase a Vendor ID from USB-IF for a one-time fee of \$5000. Note however that simply buying a VID does not authorize you to use the USB logo with a product. In order to do so, you need to become a USB-IF non-member logo licensee. This licensing fee comes at a cost of \$3,500 for a two year term. Also, be aware that this licensing fee does not entitle you to other USB-IF membership benefits, such as compliance workshops.

After you purchase a VID, you can use it across all USB devices that you produce. The Product ID is not purchased but chosen by the developer or company. Each product needs its own VID/PID combination. The USB-IF recommends each vendor set up a coordinated allocation scheme for PIDs so that different teams do not inadvertently choose the same PID for different products. Duplicate numbers will cause compliance testing to fail.

Commonly, Cypress is asked if their VID can be used on a customer's end product. The answer to this question is "No". Cypress VIDs can be used in a development environment, but cannot be released in a production design. The main reason behind this is due to operating systems, such as Windows, remember the system files (inf file, driver, and so on) used by the device and loads them every time the device is attached. If our VID is used by our customers, then the possibility that another customer may use the same VID/PID for their product exists. In this case, as you can see, the end product may bind to a different driver and malfunction. Ultimately this causes trouble for the end customer.

15 Compliance Testing

When it comes to official compliance testing for USB devices and host software, there are two options available from the USB Implementers Forum and Microsoft. This section will look at options, what the test process is from a high level, and the benefits achieved by going through the test process.

15.1 USB-IF Compliance Testing

The intention of USB is to provide a universal plug-and-play connection so that even the most non-tech savvy person can simply plug a USB cable into a host and have it working. To help ensure this, the USB IF performs compliance testing to insure that all consumers have a good overall experience with USB. As a way to provide reassurance to customers, a USB product can bear a "Certified USB" logo, similar to those in [Figure 55](#).

Figure 55. USB Compliance Logos



These logos cannot be arbitrarily put onto a product. Instead, they must undergo a series of compliance tests that are administered by USB-IF. This compliance testing is performed on products ranging from devices, hubs, host systems, silicon building blocks, and mechanical products (such as cables and connectors). These compliance tests will ensure that the device under test confirms to any relevant sections of the USB specification. Upon passing, the product will receive the ability to use the relevant USB “Certified” logo, and in addition, have the product put on the USB integrators’ list, which is simply a list maintained by USB-IF that contains the list of USB products that have met the mandatory compliance criteria. There are two components that must be completed on the path to being certified as USB compliant: checklists and compliance testing.

15.1.1 USB-IF Compliance Checklists

There are several checklists depending on what is being certified. In essence, the checklists are a questionnaire about the device’s specifications, functionality, and behavior. Many Cypress devices will be used in a peripheral application, which means the checklists will ask varying questions regarding the mechanical design and layout, device states and signals, operating voltage, and power consumption. Each question in a checklist is accompanied with a section number that can be referenced to the USB specification itself for a more detailed explanation. These checklists act as the first step towards compliance certification.

15.1.2 USB-IF Compliance Testing

After all the checklist requirements are met, the next step is to go through the second certification criteria: compliance testing. There are two ways to perform compliance testing on a device. Companies that are a member of USB-IF may attend a compliance testing workshop (also known as a Plugfest). The testing can also be contracted out to an independent certified lab. The following link is a list of [Independent Test Labs](#) provided by USB-IF. As mentioned earlier, you must be a member of USB-IF to attend the plugfest workshops. There are advantages and disadvantages to each method.

Table 13. Comparing Test Labs Versus Plugfests

| Independent Test Lab | USB-IF Plugfest |
|---------------------------------------------------|------------------------------------------------------------------------------------------|
| No personal required to be present during testing | One to four days of an engineer’s or technician’s time is required to attend the event |
| ~\$2K-6K cost per item | Free with USB-IF membership (excluding time, travel, and so on) |
| Consulting available for a fee | USB experts available to help with debug, but resources are shared among other attendees |
| Testing can occur at any time | Testing events only held four or five times a year |
| Exposed to less prototype hardware. | Exposed to prototype hosts, hubs, and drivers |
| More secrecy | Less secrecy |

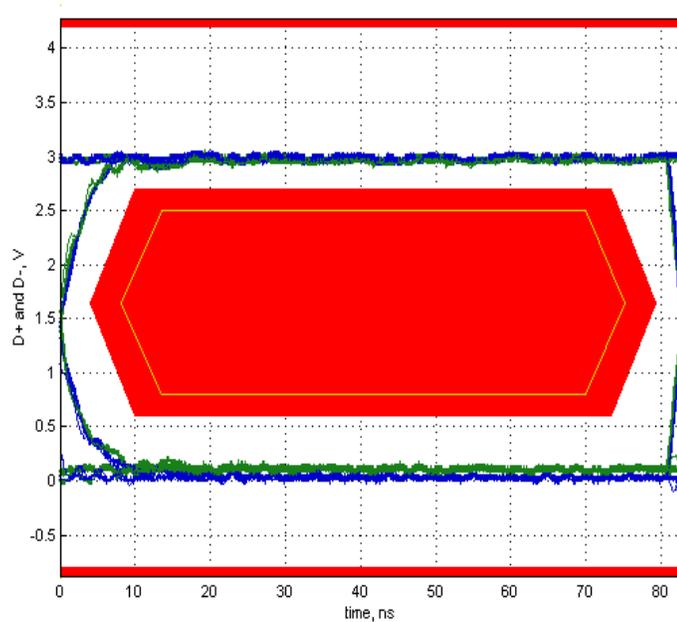
Regardless of which testing method is chosen, the checklists completed earlier will be submitted officially at this point in time. The compliance testing process will differ based on the product that is being certified. These requirements can be found on the USB-IF website. In the instance of a Full-Speed device, such as PSoC, the following must be completed to earn certification: Electrical Tests, Interoperability Tests, and Functional Tests, all of which are described in the following sections.

Functional Tests: Also referred to as “Device Framework Tests”, these tests evaluate a series of items such as stress testing the device, Chapter 9 compliance tests, specific USB class tests, and general ability to function after a suspend condition. The primary tool used in this testing is called USBCV, which is discussed further in the [Preparing for Certification](#) section. Additionally, in the instance of the device being a HID, another set of tests are performed to ensure compliance to the USB-IF HID specification.

Electrical Tests: These tests check the device’s compliance to the USB electrical specification by evaluating the signal quality, ensuring the device does not cause back voltage on the bus, and that suspend/resume/reset commands are responded to within the time limits documented in the USB specification.

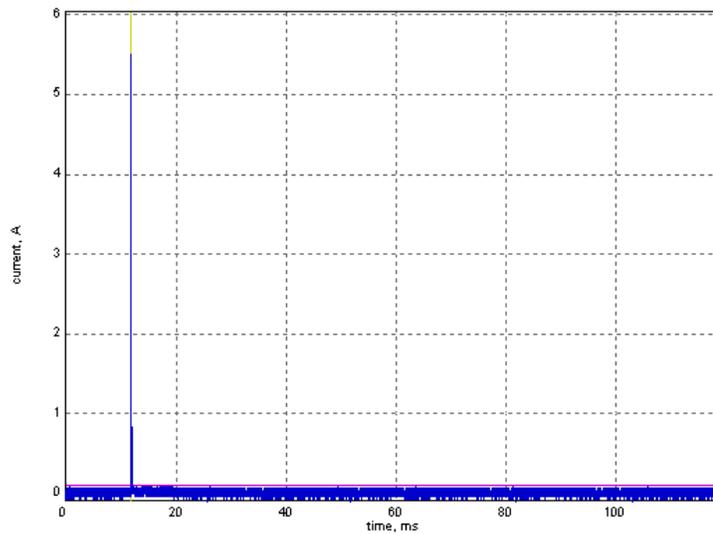
The more commonly recognized result of the electrical testing is the USB eye diagram as shown in [Figure 56](#). This diagram is created on certain high-end oscilloscopes and show the signal quality of a device by testing the rise time, fall time, undershoot, overshoot, and D+ and D- line jitter.

Figure 56. USB Eye Test Diagram



During the electrical tests, an oscilloscope captures the inrush current into the device, as shown in Figure 57. The purpose of the test is to ensure that an excessive amount of current is not drawn from the hub, and if so, to ensure that the proper current surge limiting circuitry exists.

Figure 57. USB In-Rush Current Measurement



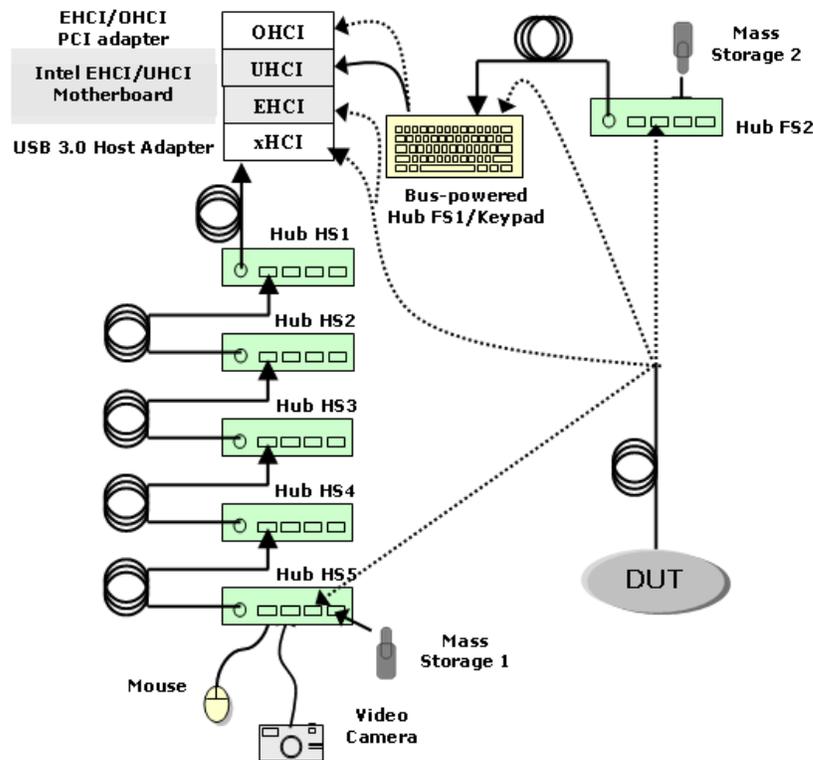
Performing the electrical testing uses a series of different adapter boards and test hardware. Particularly, testing is performed using high-end oscilloscopes. USB-IF provides a list of compliance test procedures, depending on the oscilloscope used, on their website.

Interoperability Tests: These tests evaluate the device’s ability to interact with a host and coexist with other USB devices. A device can play with many combinations of devices, hosts, and hubs. Various procedures are available for interoperability depending on whether the device is a **Low-Speed, Full-Speed, High-Speed, or SuperSpeed** device. There is a term that is often heard in the USB community that is associated with interoperability. That term is “Gold Tree” and is an arrangement of USB peripherals, for use in testing that consists of the following characteristics:

- The bus contains isochronous, bulk, interrupt, and control traffic.
- The bus contains both Full-Speed and High-Speed traffic.
- The device is nested behind five levels of USB hubs.
- The device is up to 30 meters from the host.
- The host contains EHCI, UHCI, and OHCI controllers for testing.

To create the High-Speed and Full-Speed bus traffic, discussed in bullet point 2, the interoperability test also defines what test devices are to be included in the gold tree. The interoperability document even gives specifics on the manufacturer and model of the hardware to be used in testing. From a high level, the attached hardware includes a video camera (such as a web camera), a mass storage device (such as a USB hard drive), a flash media drive (Flash Drive or Jump Drive), a keyboard, and a mouse. All of this hardware is pieced together to create the web shown in the following figure.

Figure 58. USB Gold Tree Attachment Points



The process of interoperability testing consists of multiple steps.

- Step 1:** Hot Detach & Reattach: Detach the device under test and reattach it to the same hub port.
- Step 2:** Topology Change - Detach the device under test and attach it to a different port location.
- Step 3:** Warm Boot - Restart the host/system through the Windows Start Menu (Start > Shutdown > Restart)
- Step 4:** Cold Boot – Restart the host/system through the Windows Start Menu (Start > Shutdown > Shutdown). Restart the PC by using the physical power button on the PC.

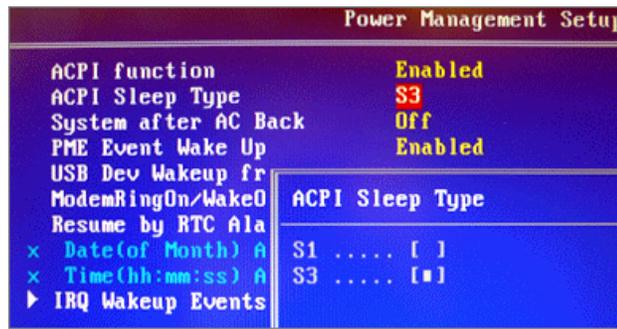
Note: The reason to perform a cold boot and a warm boot is that a warm boot typically does not go through the complete boot process. Often, the Power-On Self-Test (POST) will be skipped. In addition, a warm boot may not reset all onboard devices.

The remaining tests of interoperability occur in various system power states, solely with regards to sleeping states. To be more specific, interoperability tests S1 and S3 states.

- S0: Working State
- S1/S2: CPU Stopped
- S3: Suspend to RAM
- S4: Suspend to Disk (known as Hibernate)

Microsoft provides a more detailed description of what exactly the PC is doing in these various states in their [System Sleeping States](#) page. These states are entered when the PC is put into Sleep Mode. However, deciding whether the PC goes into S1 or S3 is determined by the BIOS for the host PC. The ACPI sleep type is typically found in the Power Management Setup of the BIOS itself, as shown in [Figure 59](#). Thus, between the S1 and S3 tests, which will be described shortly, the test PC needs to be booted, entered into the BIOS configuration menu, and have the ACPI sleep mode configured properly as shown in [Figure 59](#).

Figure 59. BIOS Selection of ACPI Sleep Mode



Step 5: Active S1 Suspend - With the BIOS configured for S1 mode, place the system into a sleep state. Note that the device should be in an active state prior to initiating the suspend state. Wake the system and confirm that the operation initiated prior to suspend continues without error. If the device being tested supports Remote Wakeup, then that approach should be used to resume the system.

Step 6: Inactive S1 Suspend - With the BIOS still configured for S1 mode, place the system into a sleep state while the device is in an idle state. Wake the system and confirm that the device under test still functions as intended when commanded. If the device being tested supports Remote Wakeup, then that approach should be used to resume the system.

Step 7: Active S3 Suspend - Configure the BIOS for S3 mode and place the system into a sleep state, similar to Step 5. Note that the device should be in an active state prior to initiating the suspend state. Wake the system and confirm that the device under test still functions as intended. Remote wakeup should be disabled during this test.

When testing is complete and checklists submitted, in a matter of a couple weeks, you can find a device listed on the USB-IF Integrators List. The following image provides an example of a listed device. In this case, the tested PSoC 3 can be seen.

Figure 60. PSoC 3 Device Shown on Integrators List

| Name | Company | TID | Categories | Added |
|-----------------------------------------------------------|--------------------------|----------|--------------------------------------------------------------|-------------------------|
| PSoC 3 CY8C3866AXI-040 | Cypress Semiconductor | 40770053 | Development > Peripheral Silicon > Low/Full Speed > Other | 29 Apr 2011 00:31:00 |

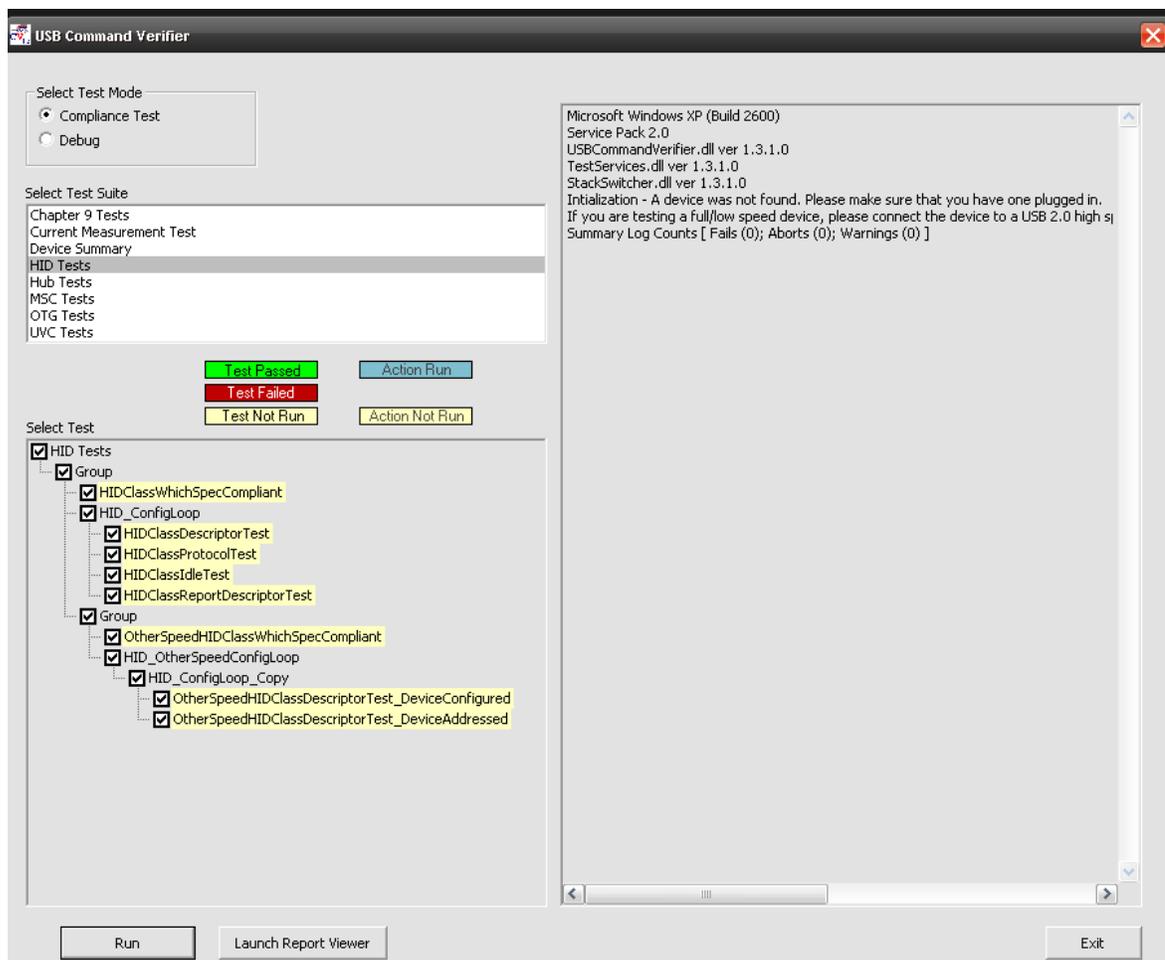
In the event that a device exceeded a specification parameter, there is still a way to pass compliance testing overall. That way is through the use of waivers. Waivers allow devices to qualify for the Integrator's List when they are slightly out of spec. For example, the current High-Speed test procedure allows devices to pass the High-Speed electrical test if they fail in the fifth hub tier. Waivers are temporary agreements between the device manufacturer and the USB I/F. As the current practices of USB design improve, waivers are removed. Waiver decisions are made by the Compliance Review Board.

15.1.3 Preparing for Certification

Before you have a device officially certified, there are multiple applications provided by USB-IF that can be used to test the device compliance. These applications are located in the [Tools section](#) of their website. Two of the more commonly used devices with USB peripherals are USBCV and USBET. Both these tools are available as free downloads from the USB-IF website.

USBCV is a command-verifier application that is used to test the compliance with Chapter 9 of the USB specification (Device Framework) and tests the compliance with class specifications (such as HID). [Figure 61](#) is a screen capture of USBCV.

Figure 61. USBCV Application



USBET and **USBHSET** are tools that test electrical properties of a device. This includes the ability to test inrush and suspend mode current, and D+/D- signal quality. **USBHSET** tests the electrical characteristics of a high-speed device, and is an excellent tool for sending certain device commands. You do not need a high-speed device when using **USBHSET**. Not having that dependency makes this application extremely useful for testing full-speed devices. [Figure 62](#) and [Figure 63](#) show a screen capture of these applications. There are also additional tools called **USBHTT**, **OPT**, and **OET**, which are intended for hubs and USB On-The-Go devices.

Figure 62. USBET Application

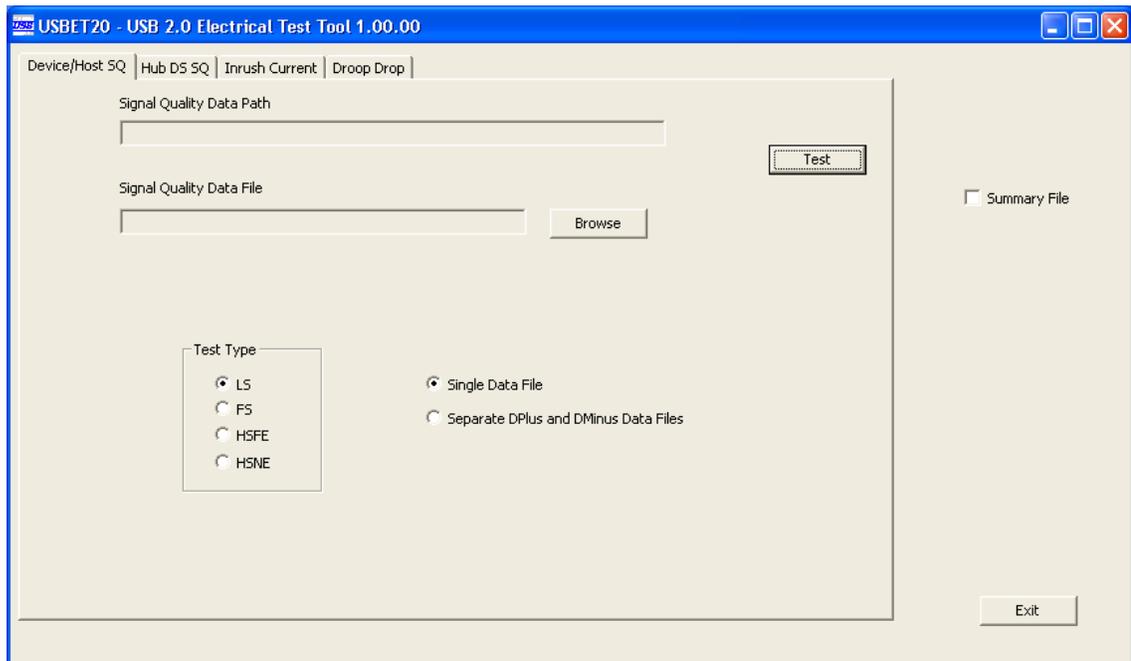
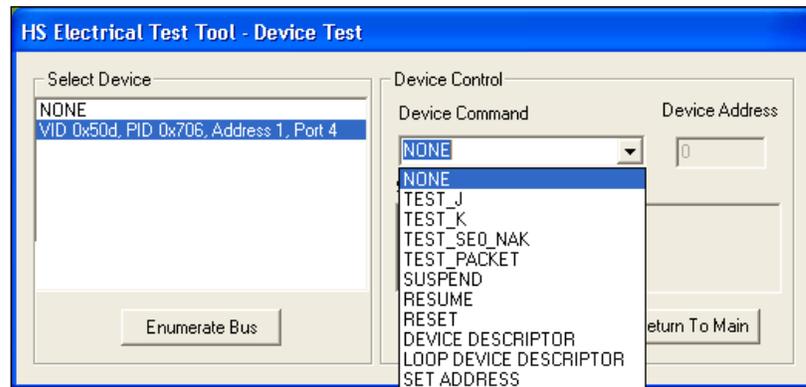


Figure 63. USBHSET Application



15.1.4 Qualification by Similarity

One question that might be on your mind is with regards to having different products that are extremely similar. What happens to them? USB-IF has a special circumstance for that called Qualification by Similarity, of which the details from the USB-IF website is as follows:

When products are very similar, testing of one product may also allow other products to be added to the Integrators List. Many OEMs buy USB interface boards that are already on the Integrators List and qualify by similarity.

However, if 'significant differences' exist between products, testing of each is required. The definition of 'significant differences' is debatable and the final judgment is the responsibility of the compliance review board, which reports to the USB-IF board of directors. As decisions are made on what are 'significant differences', rules of thumb are listed at <http://www.usb.org/developers/compliance/>.

The ultimate responsibility for making sure that various production product models do not have 'significant differences' from the product samples tested lies with each vendor. Audits by USB-IF that reveal discrepancies between shipping product and samples tested are cause for retest. The effect on rights to use the USB-I/F logo is covered in the standard logo license agreement.

Retest required:

- Microcontroller design change (new architecture or new product family).
- Connector footprint on PCB.

Retest not required:

- Product packaging changes (color, shape, and so on)
- Microcontroller vendor change (no board layout change, no firmware change). Retest not required only if new microcontroller is on the Integrators List.
- Microcontroller firmware change (changes in fully modular code not associated with USB functions).
- Connector color and aesthetics.

15.2 Microsoft Hardware Certification Testing

When you connect a USB device into Microsoft Windows computer, you often receive a warning message that the device does not have a digitally signed driver. These warnings are similar to those seen in [Figure 64](#).

Figure 64. Windows 7 Driver Warning



These warnings are a result of device developers not submitting their device for Windows Hardware Quality Labs testing, also known as WHQL or Windows Logo testing. WHQL testing is Microsoft's test plan for USB hardware or software to verify that the device does not cause compatibility issues with Windows that may cause Windows to crash or fail to function properly. The program covers the certification for the following Windows platforms: Windows 8/8.1, Windows 7, Windows Vista, Windows Server 2012/2012 R2, and Windows Server 2008/2008R2.

When a logo testing is performed on a device or driver, it goes through a specific series of tests that are tailored to that specific device class, such as a printer or a touch screen. Devices that do not fit into Microsoft's defined set of devices can still submit a driver as "Unclassified" to obtain a digital signal. This will remove the error in [Figure 64](#), but the product cannot display the Windows logo.

Testing can be performed in one of two ways: either in-house by the company developing the product or by a third party. Microsoft does not perform the testing for you. If testing in-house, the process begins by downloading the Hardware Certification Kit (HCK) from Microsoft's website. This kit provides all the needed software, driver, and tools to tests and bundle the results to be sent to Microsoft. Note that this is a free download from Microsoft. After the toolkit is downloaded, use the installed software to perform the needed tests and put together a submission bundle that can be given to Microsoft. Note that the test environment needed to perform the testing will require a dedicated PC with a copy of Windows Server that is responsible for administrating and controlling the tests. Another PC with the Windows operating system you intend to test against is also required. You will need to test against both 32-bit and 64-bit versions of the Windows software. Additionally, at the time of this writing, Microsoft charges \$250 per Windows family. The HCK will provide a guide that walks you through the test configuration and required tests.

The result of WHQL testing, if all tests are passed, is to create a submission bundle using the HCK that will be given to Microsoft. After Microsoft receives and accepts the submission, you will receive a signed certification file from Microsoft that is included with the driver and does not cause these warning messages. Passing WHQL testing allows devices to bear some of the following logos in [Figure 65](#) depending on the device and the OS that was tested.

Figure 65. Microsoft Windows Certified Logos



The need for WHQL testing has increased in the last few years with the release of 64-bit Windows Vista and 64-bit Windows 7, which require all drivers installed in the 64-bit system to be signed unless you enter a special startup mode that disables the enforcement. More information on Windows Hardware Certification and the process is available on Microsoft's website [here](#).

16 Summary

At this point, you should have a foundation to understand other USB material with more confidence. While this application note contains some theory, several other Cypress USB application notes show you how to use the information taught in this application note. See the [Related Resources](#) section for some of those resources. Additional information regarding anything that was mentioned in this application note is also contained in the USB specification. I strongly recommend that for anything you wish to explore and learn about, you begin there.

17 Related Resources

Application Notes

- [AN57473](#) - USB HID Basics with PSoC 3 and PSoC 5LP
- [AN58726](#) - USB HID Intermediate with PSoC 3 and PSoC 5LP
- [AN82072](#) - PSoC 3 and PSoC 5LP USB General Data Transfer with Standard HID Drivers
- [AN56377](#) - PSoC 3 and PSoC 5LP – Introduction to Implementing USB Data Transfers
- [AN73503](#) - USB HID Bootloader for PSoC 3 and PSoC 5LP
- [AN75705](#) – Getting Started with EZ-USB FX3
- [AN76348](#) – Differences in Implementation of EZ-USB FX2LP and EZ-USB FX3 Applications
- [Application Notes on USB Hi-Speed devices](#)

Knowledge Base Articles

Cypress gets many queries on the various USB-capable devices. These queries along with their responses are documented as knowledge base articles (KBA) on the Cypress website and are available to other customers. The following are hyperlinks to KBAs for some of our popular USB capable devices.

- [Knowledge Base Articles on PSoC devices with USB feature](#)
- [Knowledge Base Articles on dedicated USB Controllers](#)

Code Examples

Although this application note gives only a theoretical introduction to USB 2.0, Cypress provides numerous code examples for a practical understanding of the different aspects of USB, and enables you to develop your own USB-based projects. These code examples are available at the following links.

- [PSoC 3, PSoC 4, and PSoC 5LP code examples](#) – This link contains the code examples applicable for the PSoC 3, PSoC 4, and PSoC 5LP devices. Search for the “USB” keyword in this webpage to find the code examples related to USB.
- [USB Full-Speed and Low-Speed code examples](#) – This link contains the code examples applicable for the dedicated low-speed and full-speed USB devices from Cypress.
- [USB Hi-Speed code examples](#) – This link contains the code examples applicable for the hi-speed USB devices from Cypress.

Technical Support

Should you have any questions that this application note or any of the other related application notes cannot help with or require support with your design, please file a [technical support case](#).

Additional Information

- [Official USB 2.0 Specification](#)
- [USB Complete by Jan Axelson](#)

About the Author

Name: Robert Murphy
Title: Systems Engineer Staff

A Appendix A

A.1 Example PSoC 3 Full-Speed USB Device Descriptors

```

/*****
Device Descriptors
*****/
uint8 CYCODE USBFS_1_DEVICE0_DESCR[] = {
/* Descriptor Length                */ 0x12u,
/* DescriptorType: DEVICE           */ 0x01u,
/* bcdUSB (ver 2.0)                 */ 0x00u, 0x02u,
/* bDeviceClass                     */ 0x00u,
/* bDeviceSubClass                  */ 0x00u,
/* bDeviceProtocol                   */ 0x00u,
/* bMaxPacketSize0                  */ 0x08u,
/* idVendor                          */ 0xB4u, 0x04u,
/* idProduct                        */ 0x34u, 0x12u,
/* bcdDevice                        */ 0x00u, 0x00u,
/* iManufacturer                    */ 0x01u,
/* iProduct                          */ 0x02u,
/* iSerialNumber                     */ 0x00u,
/* bNumConfigurations                */ 0x01u
};
/*****
Config Descriptor
*****/
uint8 CYCODE USBFS_1_DEVICE0_CONFIGURATION0_DESCR[] = {
/* Config Descriptor Length          */ 0x09u,
/* DescriptorType: CONFIG            */ 0x02u,
/* wTotalLength                       */ 0x19u, 0x00u,
/* bNumInterfaces                    */ 0x01u,
/* bConfigurationValue                */ 0x01u,
/* iConfiguration                    */ 0x00u,
/* bmAttributes                       */ 0x80u,
/* bMaxPower                          */ 0x32u,
/*****
Interface Descriptor
*****/
/* Interface Descriptor Length        */ 0x09u,
/* DescriptorType: INTERFACE          */ 0x04u,
/* bInterfaceNumber                   */ 0x00u,
/* bAlternateSetting                   */ 0x00u,
/* bNumEndpoints                      */ 0x01u,
/* bInterfaceClass                     */ 0xFFu,
/* bInterfaceSubClass                 */ 0x00u,
/* bInterfaceProtocol                 */ 0x00u,
/* iInterface                          */ 0x00u,
/*****
Endpoint Descriptor
*****/
/* Endpoint Descriptor Length          */ 0x07u,
/* DescriptorType: ENDPOINT           */ 0x05u,
/* bEndpointAddress                   */ 0x81u,
/* bmAttributes                       */ 0x02u,
/* wMaxPacketSize                     */ 0x40u, 0x00u,
/* bInterval                          */ 0x00u
};

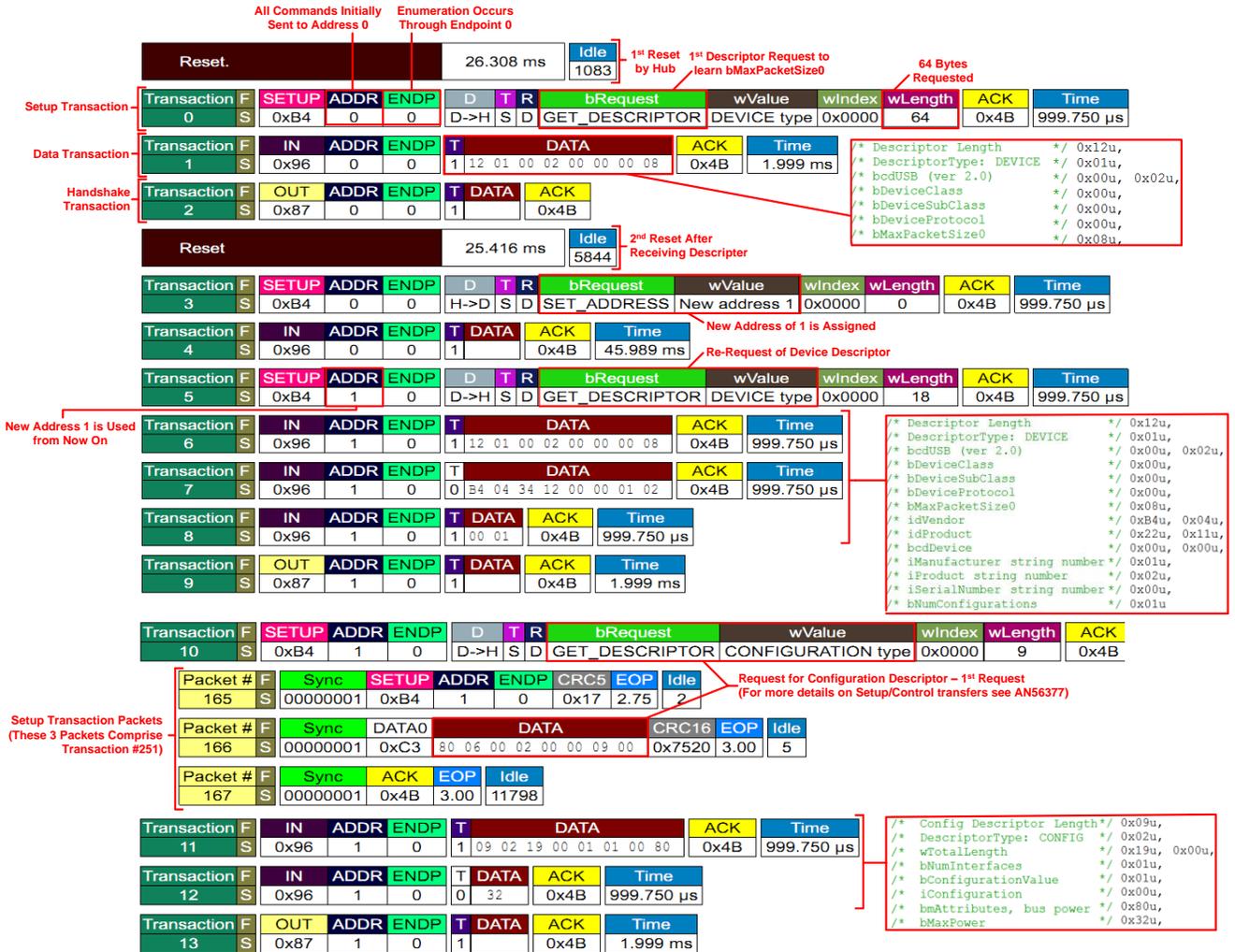
/*****
String Descriptor Table
*****/

```

```
uint8 CYCODE USBFS_1_STRING_DESCRIPTOR[] = {
/*****
Language ID Descriptor
*****/
/* Descriptor Length          */ 0x04u,
/* DescriptorType: STRING    */ 0x03u,
/* Language Id               */ 0x09u, 0x04u,
/*****
String Descriptor: "Cypress Semiconductor"
*****/
/* Descriptor Length          */ 0x2Cu,
/* DescriptorType: STRING    */ 0x03u,
/* Character array: 'C', 0, 'y', 0, 'p', 0, 'r', 0, 'e', 0, 's', 0, 's', 0, 'i', 0, 's', 0, 'e', 0,
/* Character array: 'm', 0, 'i', 0, 'c', 0, 'o', 0, 'n', 0, 'd', 0, 'u', 0, 'c', 0, 't', 0, 'o', 0,
/* Character array: 'r', 0,
/*****
String Descriptor: "USB Example"
*****/
/* Descriptor Length          */ 0x18u,
/* DescriptorType: STRING    */ 0x03u,
/* Character array: 'U', 0, 'S', 0, 'B', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0, 'm', 0, 'p', 0, 'l', 0,
/* Character array: 'e', 0,
/*****
/* Marks the end of the list.          */ 0x00u};
/*****/0,
```

B Appendix B

B.1 Bus Analyzer Capture of USB Enumeration (Example)



Request for Configuration (2nd Request), Interface, and Endpoint Descriptors

| | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|---|---|----------------|--------------------|--------|---------|------|------------|
| Transaction 14 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | CONFIGURATION type | 0x0000 | 255 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 15 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 1 | 09 02 19 00 01 01 00 80 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 16 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 0 | 32 09 04 00 00 01 FF 00 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 17 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 1 | 00 00 07 05 81 02 40 00 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|------|------|----------|
| Transaction 18 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 0 | 00 | 0x4B | 1.999 ms |

| | | | | | | | | |
|----------------|---|------|------|------|---|------|------|----------|
| Transaction 19 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms |

Request for Device Qualifier Descriptor.

| | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|---|---|----------------|-----------------------|--------|---------|------|------------|
| Transaction 20 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | DEVICE_QUALIFIER type | 0x0000 | 10 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|------|------|------|----------|
| Transaction 21 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 0x78 | | 0x4B | 2.000 ms |

This is a Full Speed only device. As a result, a STALL (request error) is issued.

1st String Descriptor Request (Language IDs)

| | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|---|---|----------------|-------------------------------------|--------------------|---------|------|------------|
| Transaction 22 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | STRING type, LANGID codes requested | Language ID 0x0000 | 255 | 0x4B | 999.833 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------|------|----------|
| Transaction 23 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 1 | 04 03 09 04 | 0x4B | 1.999 ms |

| | | | | | | | | |
|----------------|---|------|------|------|---|------|------|----------|
| Transaction 24 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms |

| | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|---|---|----------------|----------------------|--------------------|---------|------|------------|
| Transaction 25 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | STRING type, Index 2 | Language ID 0x0409 | 255 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 26 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 1 | 18 03 55 00 53 00 42 00 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 27 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 0 | 20 00 45 00 78 00 61 00 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|-------------------------|------|------------|
| Transaction 28 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 1 | 6D 00 70 00 6C 00 65 00 | 0x4B | 999.750 µs |

| | | | | | | | | |
|----------------|---|------|------|------|---|------|------|----------|
| Transaction 29 | F | IN | ADDR | ENDP | T | DATA | ACK | Time |
| | S | 0x96 | 1 | 0 | 0 | | 0x4B | 1.999 ms |


```

/* Config Descriptor Length */ 0x09u,
/* DescriptorType: CONFIG */ 0x02u,
/* wTotalLength */ 0x19u, 0x00u,
/* bNumInterfaces */ 0x01u,
/* bConfigurationValue */ 0x01u,
/* iConfiguration */ 0x00u,
/* bmAttributes, bus power */ 0x80u,
/* bMaxPower */ 0x32u,
/* Interface Descriptor Length */ 0x09u,
/* DescriptorType: INTERFACE */ 0x04u,
/* bInterfaceNumber */ 0x00u,
/* bAlternateSetting */ 0x00u,
/* bNumEndpoints */ 0x01u,
/* bInterfaceClass */ 0xFFu,
/* bInterfaceSubClass */ 0x00u,
/* bInterfaceProtocol */ 0x00u,
/* iInterface */ 0x00u,
/* Endpoint Descriptor Length */ 0x07u,
/* DescriptorType: ENDPOINT */ 0x05u,
/* bEndpointAddress, EPI IN */ 0x81u,
/* bmAttributes BULK transfer type */ 0x02u,
/* wMaxPacketSize 64 bytes */ 0x40u, 0x00u,
/* bInterval */ 0x0Au
    
```



```

/* Descriptor Length */ 0x04u,
/* DescriptorType: STRING */ 0x03u,
/* Language Id */ 0x09u, 0x04u,
    
```



```

/* Descriptor Length */ 0x18u,
/* DescriptorType: STRING */ 0x03u,
'U', 0, 'S', 0, 'B', 0, ' ', 0, 'E', 0, 'x', 0, 'a', 0, 'm', 0, 'p', 0, 'l', 0,
'e', 0,
    
```


UTF16 Conversion:

U = 55h a = 61h
S = 53h m = 6Dh
B = 42h p = 70h
E = 45h l = 6Ch
x = 78h e = 65h
<Space> = 20h

Re-Request of Descriptors by Driver

| | | | | | | | | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|-------------------------|------|----------------|-------------------------------------|--------------------|---------|------|------------|--|--|--|--|--|--|--|
| Transaction 30 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms | | | | | | | | | | | | |
| Transaction 31 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time | | | | | | | |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | STRING type, LANGID codes requested | Language ID 0x0000 | 255 | 0x4B | 999.750 μs | | | | | | | |
| Transaction 32 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 04 03 09 04 | 0x4B | 1.999 ms | | | | | | | | | | | | |
| Transaction 33 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms | | | | | | | | | | | | |
| Transaction 34 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time | | | | | | | |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | STRING type, Index 2 | Language ID 0x0409 | 255 | 0x4B | 999.750 μs | | | | | | | |
| Transaction 35 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 18 03 55 00 53 00 42 00 | 0x4B | 999.833 μs | | | | | | | | | | | | |
| Transaction 36 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 0 | 20 00 45 00 78 00 61 00 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 37 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 6D 00 70 00 6C 00 65 00 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 38 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 0 | | 0x4B | 1.999 ms | | | | | | | | | | | | |
| Transaction 39 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 29.993 ms | | | | | | | | | | | | |
| Transaction 40 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time | | | | | | | |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | DEVICE type | 0x0000 | 18 | 0x4B | 999.750 μs | | | | | | | |
| Transaction 41 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 12 01 00 02 00 00 00 08 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 42 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 0 | 84 04 34 12 00 00 01 02 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 43 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 00 01 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 44 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms | | | | | | | | | | | | |
| Transaction 45 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time | | | | | | | |
| | S | 0xB4 | 1 | 0 | D->H | S | D | GET_DESCRIPTOR | CONFIGURATION type | 0x0000 | 9 | 0x4B | 999.750 μs | | | | | | | |
| Transaction 46 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 1 | 09 02 19 00 01 01 00 80 | 0x4B | 999.750 μs | | | | | | | | | | | | |
| Transaction 47 | F | IN | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x96 | 1 | 0 | 0 | 32 | 0x4B | 999.833 μs | | | | | | | | | | | | |
| Transaction 48 | F | OUT | ADDR | ENDP | T | DATA | ACK | Time | | | | | | | | | | | | |
| | S | 0x87 | 1 | 0 | 1 | | 0x4B | 1.999 ms | | | | | | | | | | | | |

Re-Request of Descriptors by Driver (Continued)

| | | | | | | | | | | | | | |
|----------------|---|-------|------|------|------|---|---|-------------------------|---------------------|-------------------------|---------|------|------------|
| Transaction 49 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | | | | D->H | S | D | GET_DESCRIPTOR | CONFIGURATION type | 0x0000 | 25 | 0x4B | 999.833 μs |
| Transaction 50 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | 09 02 19 00 01 01 00 80 | | | | 0x4B | 999.667 μs |
| Transaction 51 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 0 | | | 32 09 04 00 00 01 FF 00 | | | | 0x4B | 999.750 μs |
| Transaction 52 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | 00 00 07 05 81 02 40 00 | | | | 0x4B | 999.750 μs |
| Transaction 53 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 0 | | | 00 | | | | 0x4B | 999.750 μs |
| Transaction 54 | F | OUT | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | | | | | 0x4B | 1.999 ms |
| Transaction 55 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | | | | D->H | S | D | GET_STATUS | 0x0000 | Device Status requested | 2 | 0x4B | 999.750 μs |
| Transaction 56 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | 00 00 | | | | 0x4B | 999.750 μs |
| Transaction 57 | F | OUT | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | | | | | 0x4B | 1.999 ms |
| Transaction 58 | F | SETUP | ADDR | ENDP | D | T | R | bRequest | wValue | wIndex | wLength | ACK | Time |
| | S | | | | H->D | S | D | SET_CONFIGURATION | New configuration 1 | 0x0000 | 0 | 0x4B | 999.750 μs |
| Transaction 59 | F | IN | ADDR | ENDP | T | | | DATA | | | | ACK | Time |
| | S | | | | 1 | | | | | | | 0x4B | |

Request for Device Status

Bit 0 = Self Powered, Bit 1 = Remote Wakeup.
Both are set to zero since device is Bus Powered and Remote Wakeup is not supported.
Bits 15 through 2 are reserved and set at zero

Configuration is Selected.
Configuration 1

Device is Ready for Use!

Document History

Document Title: AN57294 - USB 101: An Introduction to Universal Serial Bus 2.0

Document Number: 001-57294

| Revision | ECN | Orig. of Change | Submission Date | Description of Change |
|----------|---------|-----------------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ** | 2796084 | BHA | 11/02/2009 | New application note. |
| *A | 3263612 | LRDK | 05/21/2011 | Rewritten in Simplified English. |
| *B | 3370282 | RLRM | 09/14/2011 | Significant content updates and additions. |
| *C | 3429310 | RLRM | 11/09/2011 | Updated template. |
| *D | 3484720 | RLRM | 01/05/2012 | Units of USB speed modified. |
| *E | 4492919 | RLRM | 09/04/2014 | Fixed typos Added the following section: Debugging USB designs Expanded the following section: Acquiring a VID and PIS, Compliance testing Expanded the Related Resources section Updated Figure 16 |
| *F | 4910757 | VVSK | 09/07/2015 | Added support for the PSoC 4200L device. Updated template. |
| *G | 5436404 | VOM | 09/14/2016 | Updated template and Related Resources section |
| *H | 5726423 | BENV | 05/04/2017 | Updated logo and copyright |

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

| | |
|-------------------------------|--------------------------------------------------------------------|
| ARM® Cortex® Microcontrollers | cypress.com/arm |
| Automotive | cypress.com/automotive |
| Clocks & Buffers | cypress.com/clocks |
| Interface | cypress.com/interface |
| Internet of Things | cypress.com/iot |
| Memory | cypress.com/memory |
| Microcontrollers | cypress.com/mcu |
| PSoC | cypress.com/psoc |
| Power Management ICs | cypress.com/pmic |
| Touch Sensing | cypress.com/touch |
| USB Controllers | cypress.com/usb |
| Wireless Connectivity | cypress.com/wireless |

PSoC® Solutions

[PSoC 1](#) | [PSoC 3](#) | [PSoC 4](#) | [PSoC 5LP](#) | [PSoC 6](#)

Cypress Developer Community

[Forums](#) | [WICED IOT Forums](#) | [Projects](#) | [Videos](#) | [Blogs](#) | [Training](#) | [Components](#)

Technical Support

cypress.com/support

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

© Cypress Semiconductor Corporation, 2009-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.