AN57294

USB 101: An Introduction to Universal Serial Bus 2.0

Author: Robert Murphy
Associated Project: No
Associated Part Family: PSoC® 1 / PSoC 3 / PSoC 5
Software Version: PSoC Designer™ or PSoC Creator™
Related Application Notes: See the Related Resource Section

Abstract

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

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.

These 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

USB 3.0 is not discussed in this application note

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 though 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 backwards compatible with USB 1.1 and 1.0. USB is currently regulated by the USB Implementers Forum (USB-IF), which is a nonprofit organization that maintains the USB documents and compliance programs.

USB Overview

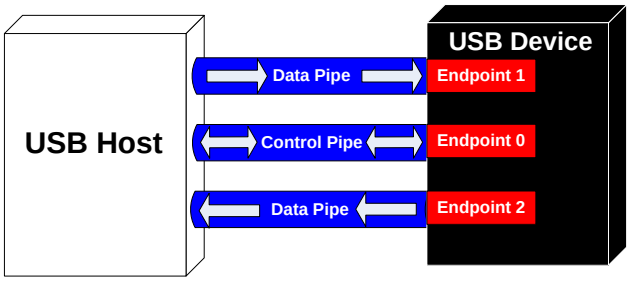
USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller 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 the 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.

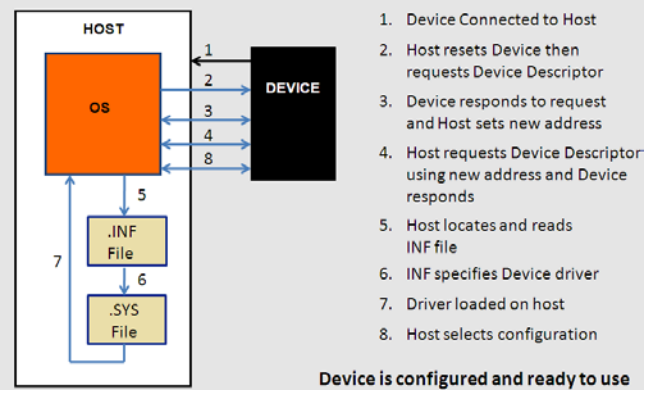
Every device has a control pipe and it is through this pipe that control transfers to send and receive messages from the device are performed. Optionally, a device may have data pipes for transferring data through interrupt, bulk, or isochronous transfers. The control pipe is the only bidirectional pipe 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 USB 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. Once this process is complete, the device is ready to transfer data to the host. The flow chart of the general enumeration process is shown is 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 has been 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.

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 then UHCI by performing more functionality in hardware.

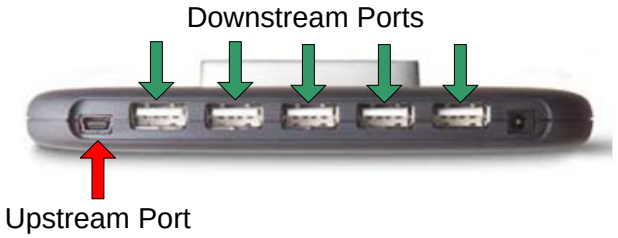
Extended Host Controller Interface (EHCI): Created for USB 2.0 after USB-IF requested that a single host controller specification be created. EHCI is used for high-speed transactions and delegates low-speed and full-speed transactions to an OHCI or UHCI sister controller.

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.

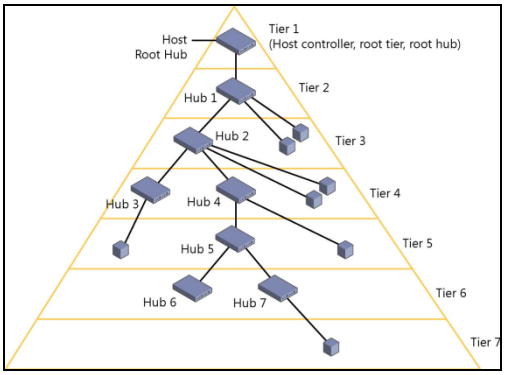
A hub is a specialized device that allows the host to communicate with multiple peripheral devices on the bus. Unlike USB peripheral devices, such as a mouse that has actual functionality, a hub device is transparent and is intended to act as a pass-through. A hub also acts as a channel between the host 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



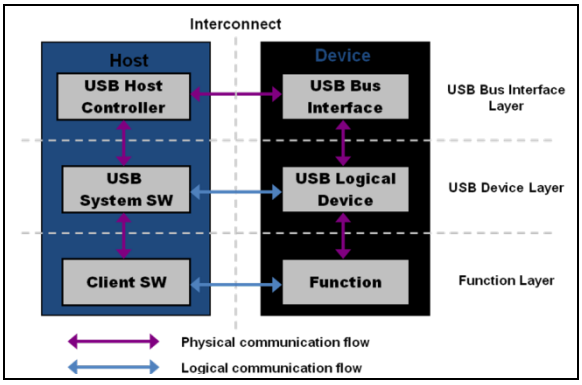
As mentioned earlier, up to 127 devices can be connected to the host controller with the use of hubs. This limitation is based on the USB protocol, which limits the device address to 7 bits. Additionally, a maximum of 5 hubs can be chained together, which is limited due to timing constraints of hub and 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 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

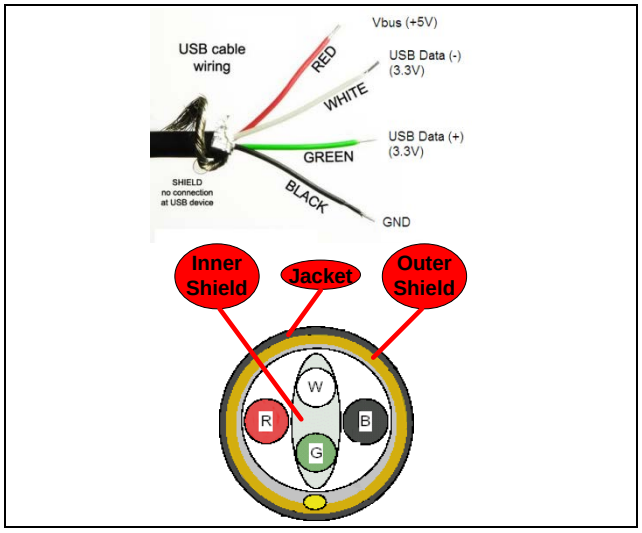


Physical Interface

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

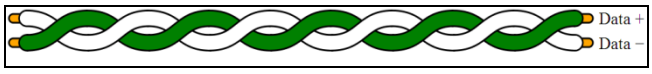
A USB cable consists of multiple components that are protected by an insulating jacket. Underneath the jacket is an outer shield that contains a copper braid. Inside the outer shield are multiple wires: a copper drain wire, a V_{BUS} wire. (red) and a ground wire (black). An inner shield made of aluminum 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.

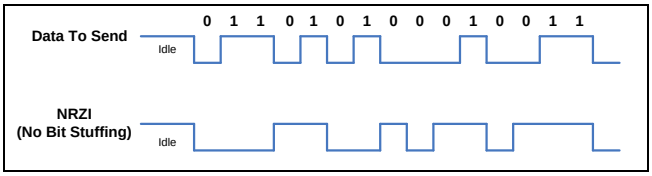
Figure 8. USB Twisted Pair Data Wires



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.

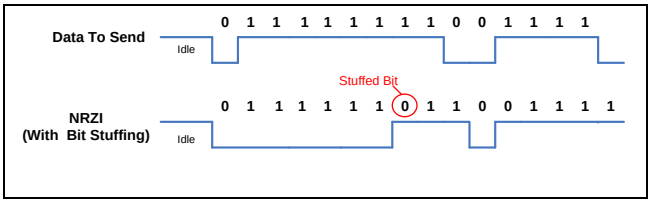
NRZI encoding is a method for mapping a binary signal for transmission over some medium. In this case a USB cable. With this encoding scheme, a logic 1 is represented by no change in voltage level and a logic 0 is represented by a change in voltage level as Figure 9 shows. On the top is the data that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

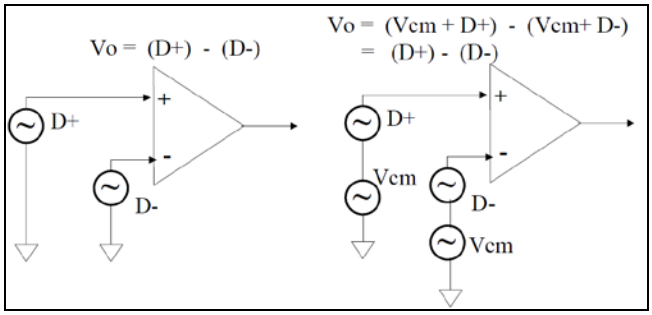
Figure 10. Data to NRZI Encoding with Bit Stuffing



The hardware in USB devices will handle all the encoding and bit stuffing upon receiving any data and prior to 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 in the 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-States and K-States. 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 vs. 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

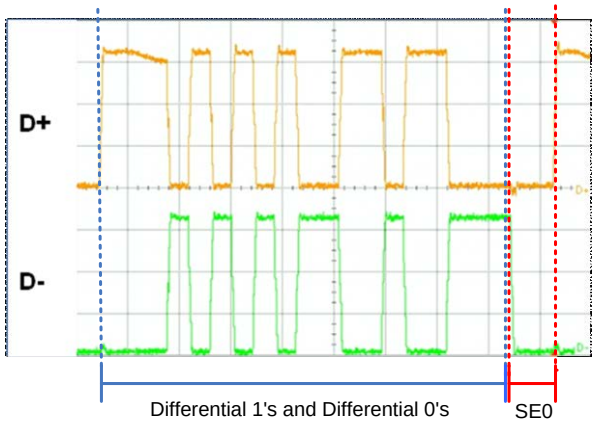


Table 1. USB Communication States

Bus State	Indication
Differential 1	D+ High, D- Low
Differential 0	D+ Low, D- High
Single Ended 0 (SE0)	D+ and D- Low
Single Needed 1 (SE1)	D+ and D- High
J-State: Low-Speed Full-Speed High-Speed	 Differential 0 Differential 1 Differential 1
K-State: Low-Speed Full-Speed High-Speed	 Differential 1 Differential 0 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 many 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

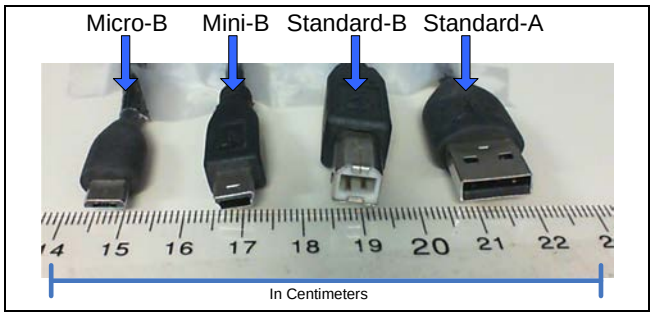


Figure 14. USB Ports and Connectors

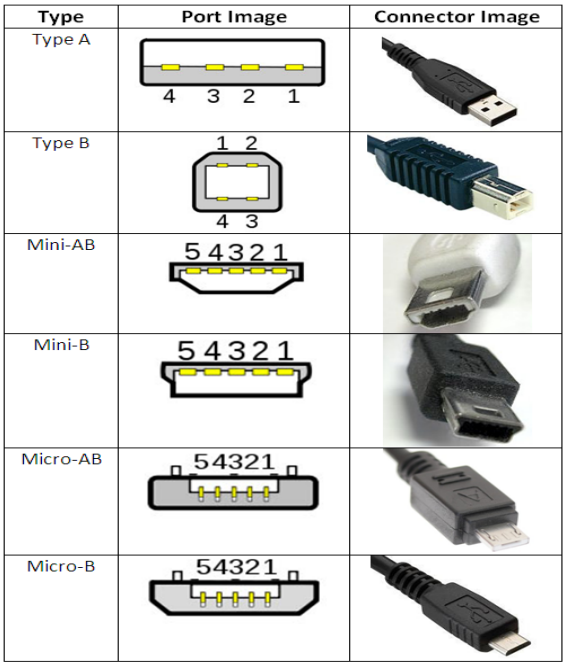


Figure 14 shows that the Mini and Micro connectors have five pins, rather than four. The extra pin is the ID pin and identifies the host and the device in OTG applications. Since PSoC does not support USB OTG, this application note does not include details about it. The reason for having different connection 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

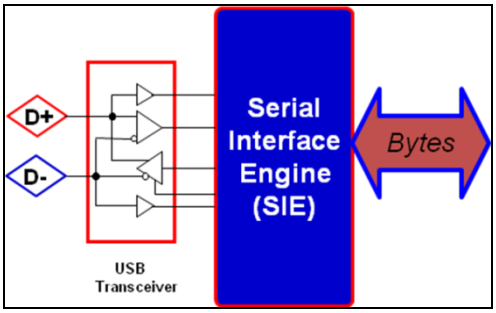
Pin	Name	Color	Function
1	V_{BUS}	Red	+5 V
2	D-	White	Data (-)
3	D+	Green	Data (+)
4	GND	Black	Ground

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.

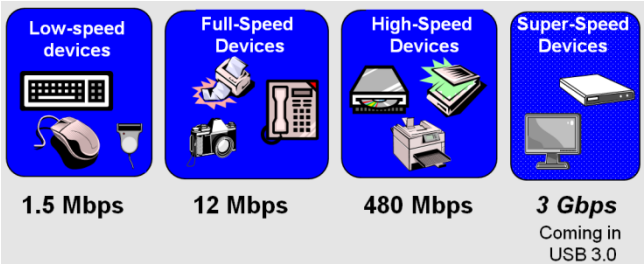
Figure 15. USB Hardware Interface at the Device



USB Speeds

Currently, the USB specification defines four speeds for a USB system: low-speed, full-speed, high-speed, and super-speed. Currently, Cypress only supports full-speed on their PSoC family of devices and low, full, and high speed on their 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.

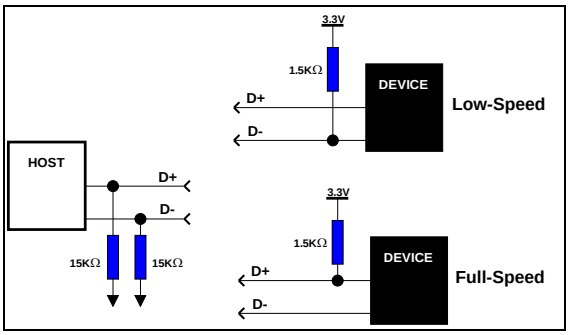
- Low-speed devices:
  - Examples: keyboards, mice, and game peripherals
  - Bus Rate: 1.5 Mb/s
  - Maximum Effective Data Rate: 800 B/s
- Full-speed devices
  - Examples: phones, audio devices, and compressed video
  - Bus Rate: 12 Mb/s
  - Maximum Effective Data Rate: 1.2 MB/s
- High-speed devices
  - 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 kΩ pull-up on the D+ line indicates that the attached device is a full-speed device. A 1.5 kΩ 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 kΩ pull-up on the D+ line. Once 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 high-speed devices are USB 2.0, but this is because high-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.

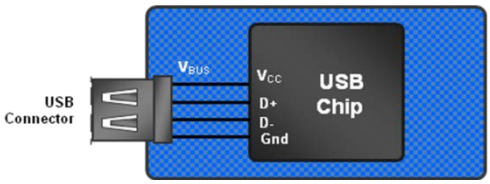
USB Power

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

Bus power is one of the many great 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

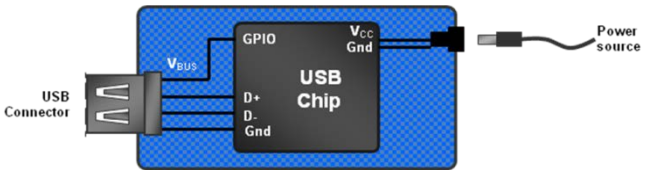
the host or a hub. When dealing with a bus-powered device, users must consider its power consumption prior to the device being put into a configured state. This is the period of time from when the device first connects to the bus, to the time that the host sends the device a SET_CONFIGURATION command 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



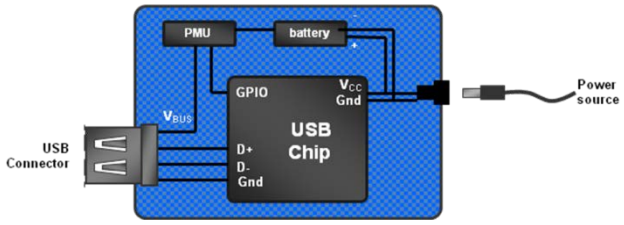
Self-powered devices supply their own power with the use of an external power source such as a DC power adapter or a battery. A self-powered device needs a different set of considerations in a design. The USB specification requires that self-powered devices monitor their V_{BUS} line at all times. A 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/or 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.

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

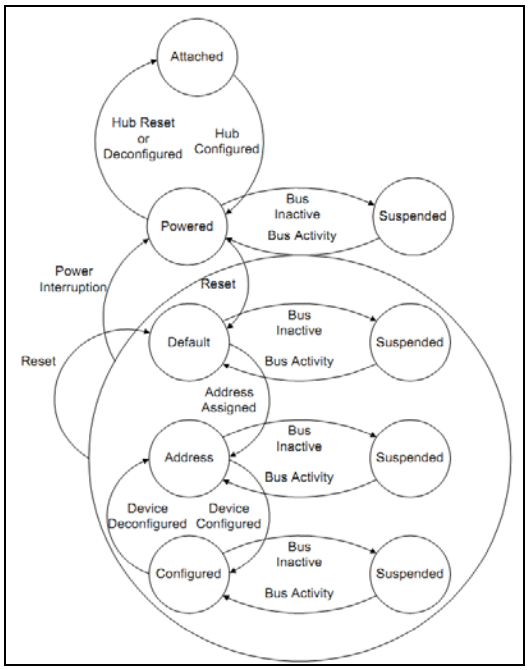
- **Attached State:** Occurs when a device is attached to a host/hub, but does not give any power to the V_{BUS} line. This is commonly seen if the hub detects an over current event. The device is still attached, but the hub removes power to it.
- **Powered:** A device is attached to the USB and has 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 below.

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.

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 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 fairly simple design such as a mouse may need only 1 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 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 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% reserved bandwidth on the bus in low and full-speed devices (20% at high-speed) and give USB system level control.

Interrupt Endpoints – These endpoint types 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 Human Interface Device (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% on low- and full-speed devices and 80% 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 byes. 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 since 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 prenegotiated bandwidth. Isochronous transfers must support streams of error tolerant data since 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 vs. 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% on low and full-speed devices (80% 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

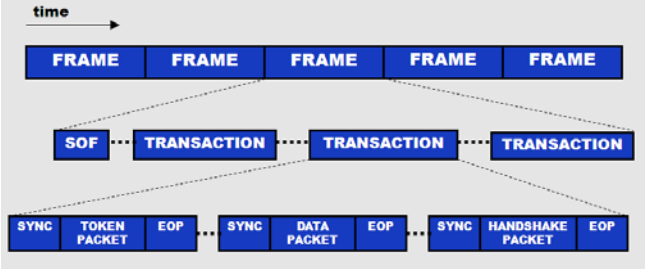
Transfer Type	Control	Interrupt	Bulk	Isochronous
Typical Use	Device Initialization and Management	Mouse and Keyboard	Printer and Mass Storage	Streaming Audio and Video
Low-speed Support	Yes	Yes	No	No
Error Correction	Yes	Yes	Yes	No
Guaranteed Delivery Rate	No	No	No	Yes
Guaranteed Bandwidth	Yes (10%)	Yes (90%)[1]	No	Yes (90%)[1]
Guaranteed Latency	No	Yes	No	Yes

[1]Shared bandwidth between isochronous and interrupt.

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.

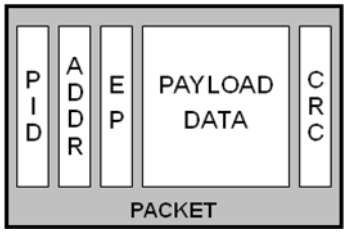
Transactions are placed within frames and are never split across frames (with the exception of high-speed isochronous transfers) or interrupt the middle of another transaction. Figure 23 shows a transaction block diagram.

Figure 23. Transaction Block Diagram

Each packet can contain different pieces of information. What information is included is dependent 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 max value of 16, we achieve 32 endpoints with an IN PID and an endpoint address between 1 -16 and a OUT PID with an endpoint address between 1 and 16, giving a total of 32. Keep in mind 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



Packet Types

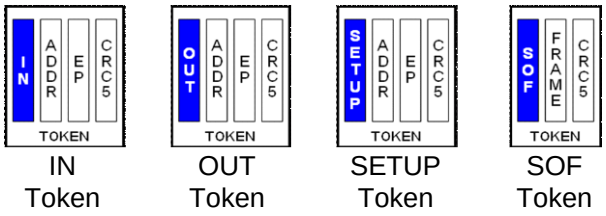
There are four different packet types that Figure 24 can potentially represent.

- Token packets
  - Initiate transaction
  - Identify device involved in transaction
  - Are 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.

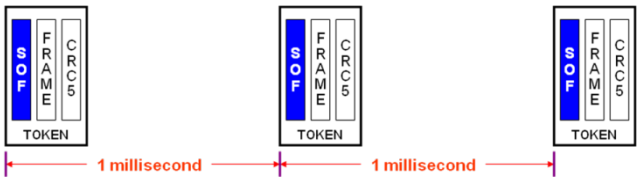
Token Packets: Token packets always come from the host and are used to direct traffic on the bus. The function of the token packet depends on the activity performed. IN tokens are used to request that devices send data to the host. OUT tokens are used to precede data from the host. SETUP tokens are used 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 below shows a diagram of the various token packets.

Figure 25. USB Token Packet Types



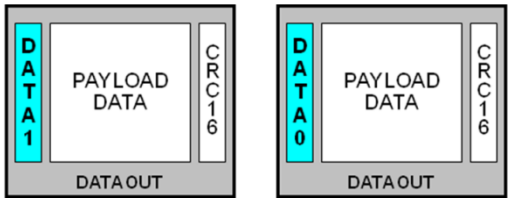
The SOF gives a way for devices to identify the beginning of a frame and synchronize with the host. They are also used to prevent a device from entering suspend mode (which it must do if 3 ms pass without an SOF). SOF packets are only seen on full and high speed devices and are sent every millisecond 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 a SOF packet. High-speed communication goes a step further with microframes. With a high-speed device, a SOF is sent out every 125 us and frame count is only incremented every 1 ms.

Figure 26. USB SOF in Full-speed Device



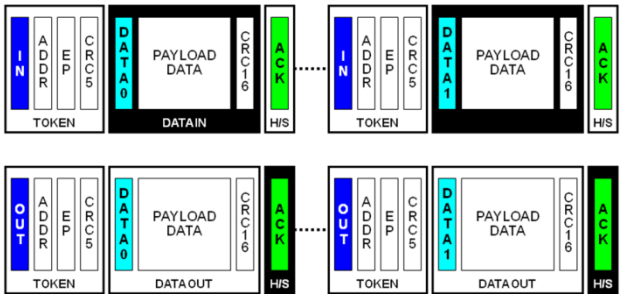
Data Packets: Data packets follow IN, OUT, and SETUP token packets. The size of the payload data ranges from 0 to 1024 bytes depending on the transfer type. The packet ID toggles between DATA0 and DATA1 for each successful data packet transfer, and the packet closes with a 16-bit CRC. The composition 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 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 Figure 28, 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)

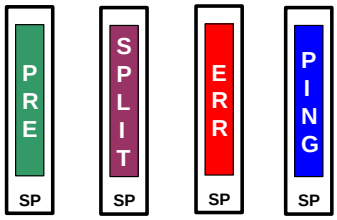
Figure 29. Handshake Packet Indicators

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



Transaction Types

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

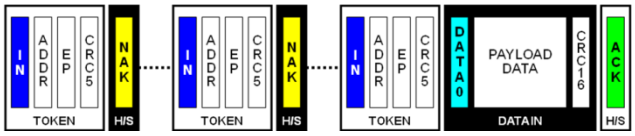
IN/Read/Upstream Transactions

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 where white boxes 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



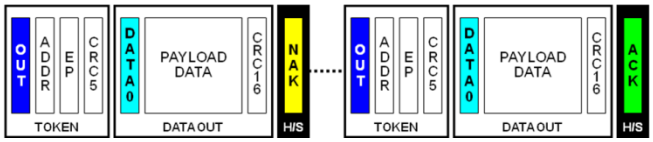
OUT/Write/Downstream Transactions

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 and 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



Control Transactions

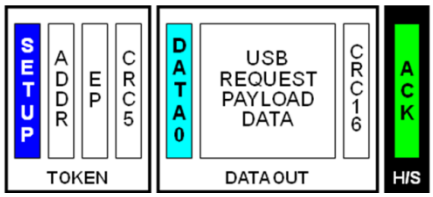
Control transfers identify, configure, and control devices. They enable the host to read information about a device, set the device address, establish configuration, and issue certain commands. A control transfer is always directed to the control endpoint of a device. Control transfers have three stages: 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 setup stage must always be

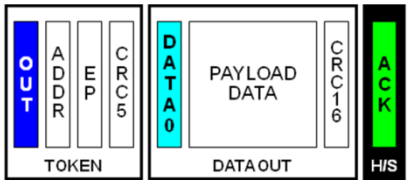
acknowledged by the device, you cannot NAK a setup stage.

Figure 36. Setup Stage Transaction



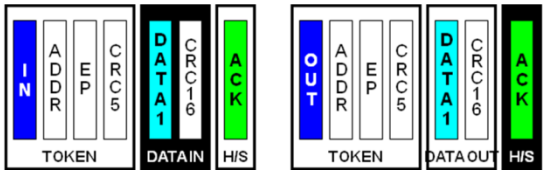
The data stage is optional in a control transaction. This stage can consist of multiple data transactions and is only required when a data payload between the host and device. Frequently, relevant data for the control transfer can be transferred in the setup stage.

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

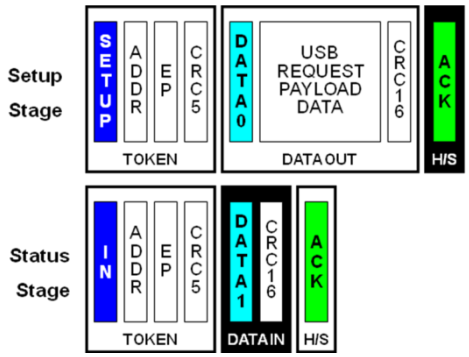


Figure 40. Example of Control Write Transaction

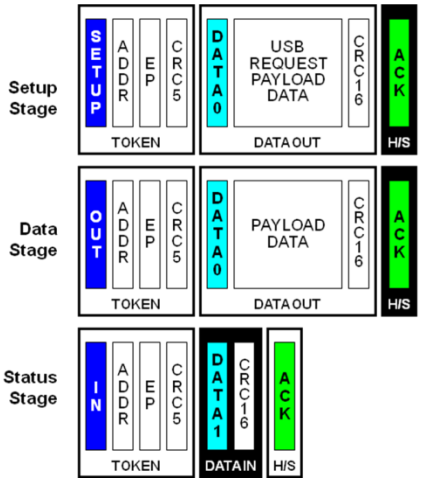
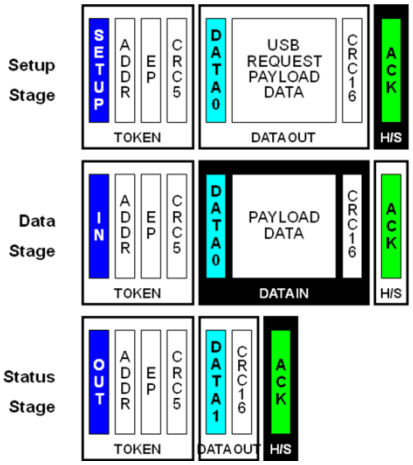


Figure 41. Example of Control Read Transaction



For more information on USB transfer types and their implementations in PSoC 3 and PSoC 5 devices, see the *AN56377 – PSoC® 3 / PSoC 5 USB Transfer Types*.

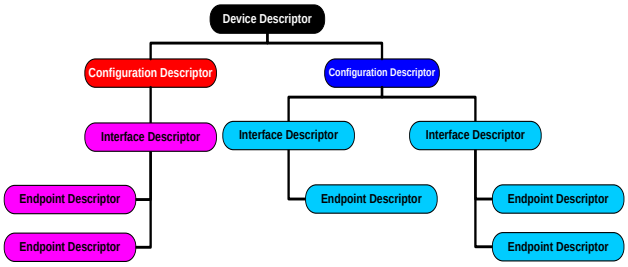
USB Descriptors

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. Those 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: Example PSoC 3 Full-Speed USB Device Descriptors 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



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 a VID, which is something that each company gets uniquely from the USB Implementers Forum), Product Identification (also known as a 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 Spec 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 class(es) in the interface descriptor, and leave these fields as 00h.

bMaxPacketSize reports the maximum number of packets supported by Endpoint zero. 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.

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.

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. The interface descriptor defines the characteristics for that 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

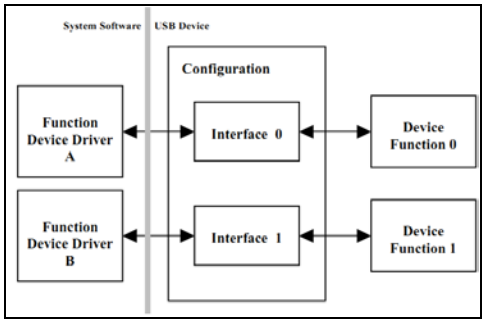


Figure 44. Multiple Interfaces with Single Function

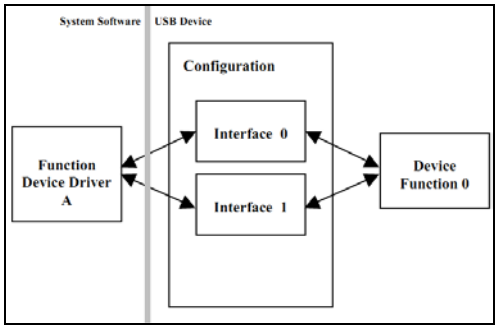


Table 7. Interface Association Descriptor Table

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

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

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)

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 stings 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.

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 - PSoC® 3 / PSoC 5 USB HID Fundamentals with Mouse and Joystick* and *AN58726 - PSoC® 3 / PSoC 5 USB HID Intermediate (with Keyboard and Composite Device)*.

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.

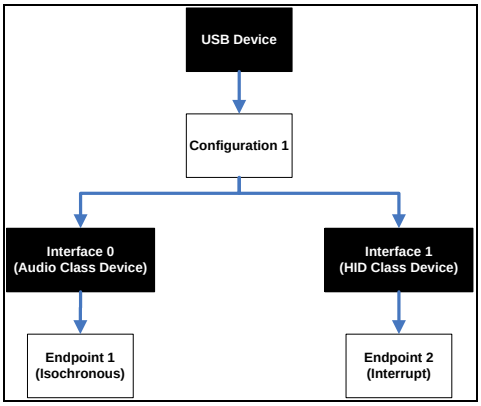
Device_Qualifier Descriptor: Another descriptor seen in USB is a Device_Qualifier Descriptor. This describes information about a high-speed capable device that changes if the device operates at another speed and is required by devices that support both speed configurations. If a device operates at full-speed 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.

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 where this might be seen is in 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 for 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 could have its endpoints configured to bulk, which has no guaranteed bus bandwidth, then in another alternate setting has the endpoints configured for Isochronous, which has guaranteed bus bandwidth. This concept is shown in Figure 46.

Figure 46. Multiple Interface Settings Diagram

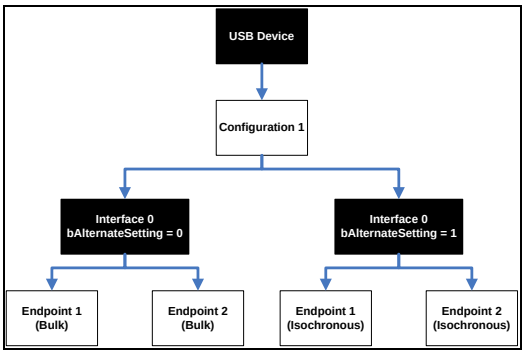
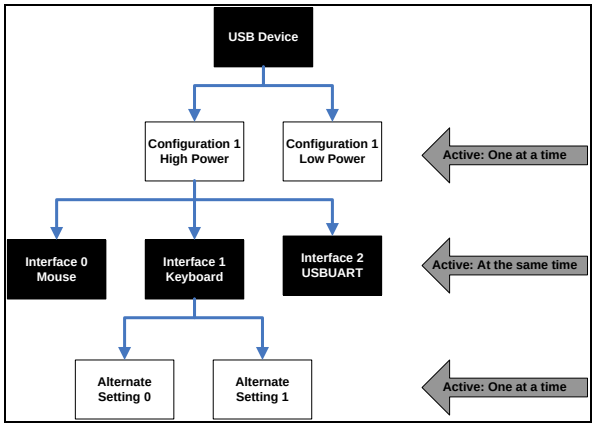


Figure 47 shows an overall diagram view of the customization options that can be used to create a highly customizable USB device to suit a variety of configurations options.

Figure 47. Configuration Diagram



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.sys
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	~80K B/s	~1MB/s	~1MB/s	~1MB/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 descriptor 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
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

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.

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.

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. Once 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 starts with this default address. Since 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 (i.e. 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 Descriptor 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 moved 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.

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. Once 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.

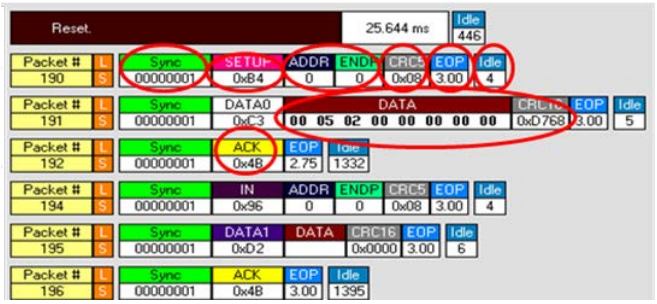
Step 13: The device is now in the configured state. It took on its proprieties that were defined with the descriptors. The defined maximum power can be drawn from V_{BUS} and the device is now ready for use in an application.

Bus Analyzer

You can see this enumeration process using a USB analyzer. A USB analyzer allows you to record bus traffic that is sent between the device and the host. They decode the USB traffic information into an easy to read display, which provides you an easy way to debug a USB design issues. Some bus analyzers allow you to generate your own USB traffic. An example of a USB bus analyzer trace is shown in Figure 48. Appendix B: Bus Analyzer Capture of USB Enumeration (Example) 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: Example PSoC 3 Full-Speed USB Device Descriptors, you have enough instruction 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 48. USB Analyzer Output Example



To help understand what Figure 48 is showing, let's 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's a SETUP token.
  - **ADDR**: The current address is zero since it's 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's 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's 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 application notes. You must understand the information that is displayed on the bus analyzer trace to fully understand the information that is being taught 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.

Acquiring a VID and PID

In order to sell or market a USB product you must have a VID. This means that you must purchase one from USBIF. It is important to make sure that you get a Vendor ID for your company to avoid legal consequences. Once 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.

USB Compliance

For a USB device to bear a "Certified USB" logo, similar to those in Figure 49, they must undergo a series of compliance tests that are administered by USB-IF.

Figure 49. USB Compliance Logos

These certification tests measure the device conformance to the USB specification. Passing these tests allows a developer to place a USB logo onto their product and have their product listed on the USB-IF Integrators List. 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.

Compliance testing consists of:

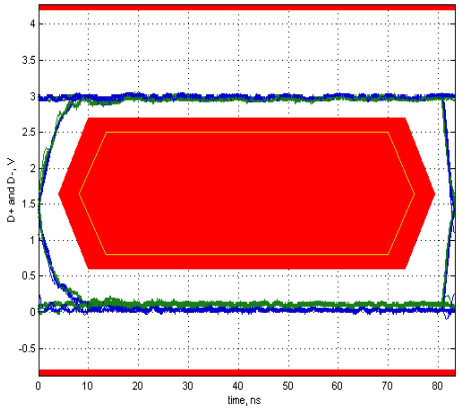
Checklist: USB-IF gives multiple checklists that ask a series of questions (derived from the USB specification) to help determine if your device is USB compliant. These checklists are completed before the USB compliance workshop and are submitted upon attendance.

Functional 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 PC reboots and returns from suspend.

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 50. 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 50. USB Eye Test Diagram

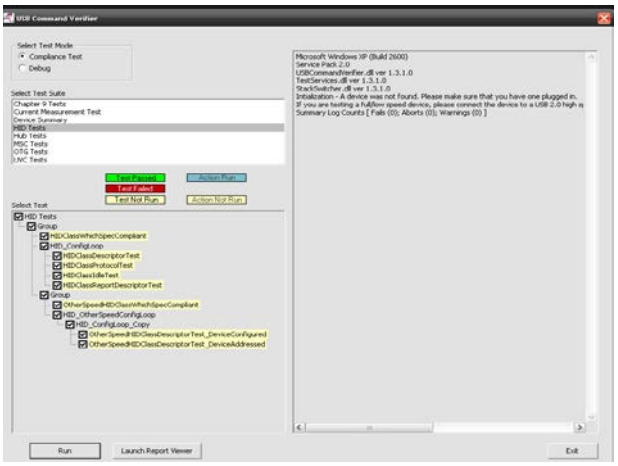


Interoperability Tests: These tests evaluate the device's ability to interact with a host and test the device's ability to coexist with other USB devices. Entering and exiting suspend mode is tested along with the current consumed in these modes.

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 compliance with Chapter 9 of the USB Specification (Device Framework) and tests compliance with class specifications (such as HID). Figure 51 is a screen capture of USBCV.

Figure 51. 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 52 and Figure 53 show a screen capture of these applications.

Figure 52. USBET Application

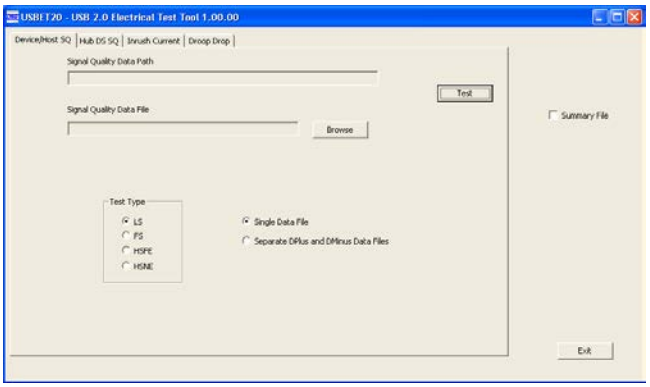
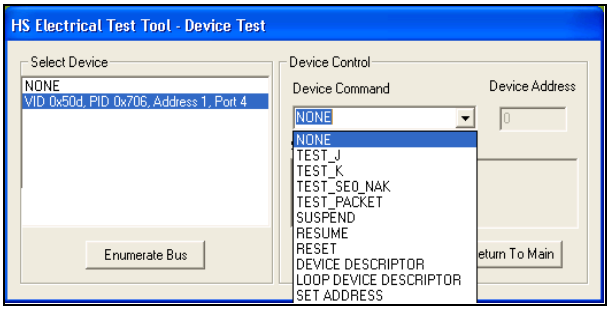


Figure 53. USBHSET Application

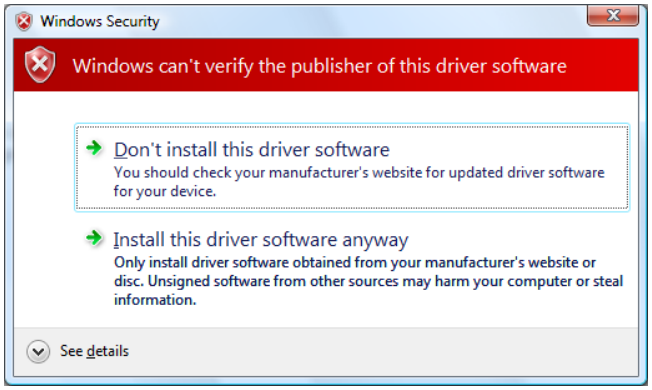


For more information regarding USB certification, read *AN023: USB Compliance Testing Overview*.

Windows Logo 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 54.

Figure 54. 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 testing. WHQL testing is Microsoft's test plan on 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 end result of WHQL testing, if all tests are passed, is to receive a signed certification file from Microsoft that is included with your diver and does not cause these warning messages. Passing WHQL testing allows devices to bear some of the following logos in Figure 55 depending on the device and the OS that was tested.

Figure 55. Microsoft Windows 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. For more information regarding Windows certification, read *AN52970: Windows Hardware Quality Lab (WHQL) Signing Procedure*.

Summary

At this point, you should have a foundation to understand other USB material with more confidence. While this application note contains some theory, there are plenty of other Cypress USB application notes that 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.

Related Resources

Application Notes:

- AN57473 - PSoC® 3 / PSoC 5 USB HID Fundamentals with Mouse and Joystick
- AN58726 - PSoC® 3 / PSoC 5 USB HID Intermediate (with Keyboard and Composite Device)
- AN56377 - PSoC® 3 / PSoC 5 USB Transfer Types
- AN023 - USB Compliance Testing Overview
- AN52970 - Windows Hardware Quality Lab (WHQL) Signing Procedure
- AN14557 – Introduction to CYUSB.dll Based Application Development Using C#
- AN61744 – Introduction to CYAPI.lib Based Application Development Using VC++

Additional Information:

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

About the Author

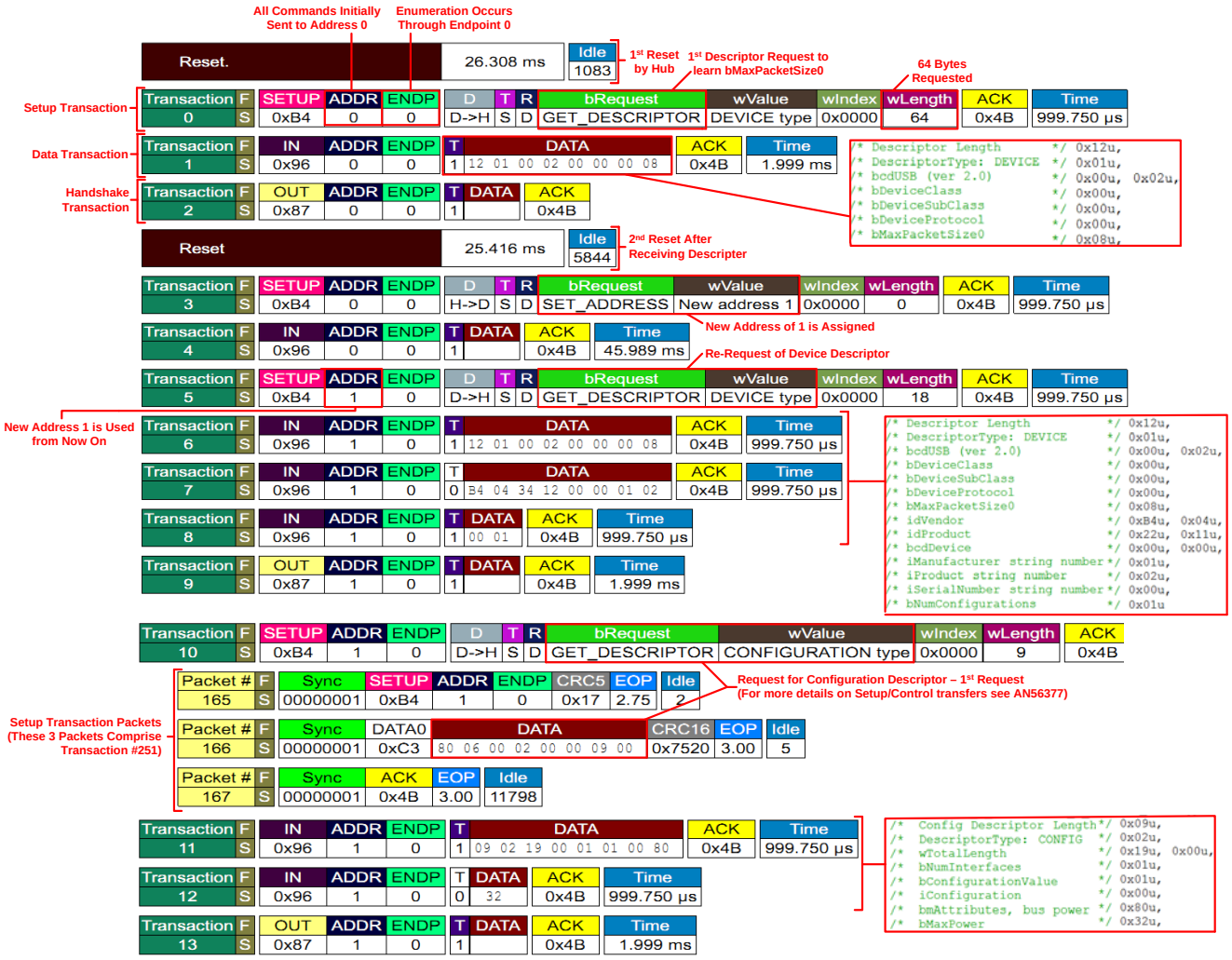
Name:	Robert Murphy
Title:	Application Engineer Sr.
Background:	Robert Murphy graduated from Purdue University with a Bachelor's Degree in Electrical Engineering Technology.
Contact:	rlrm@cypress.com

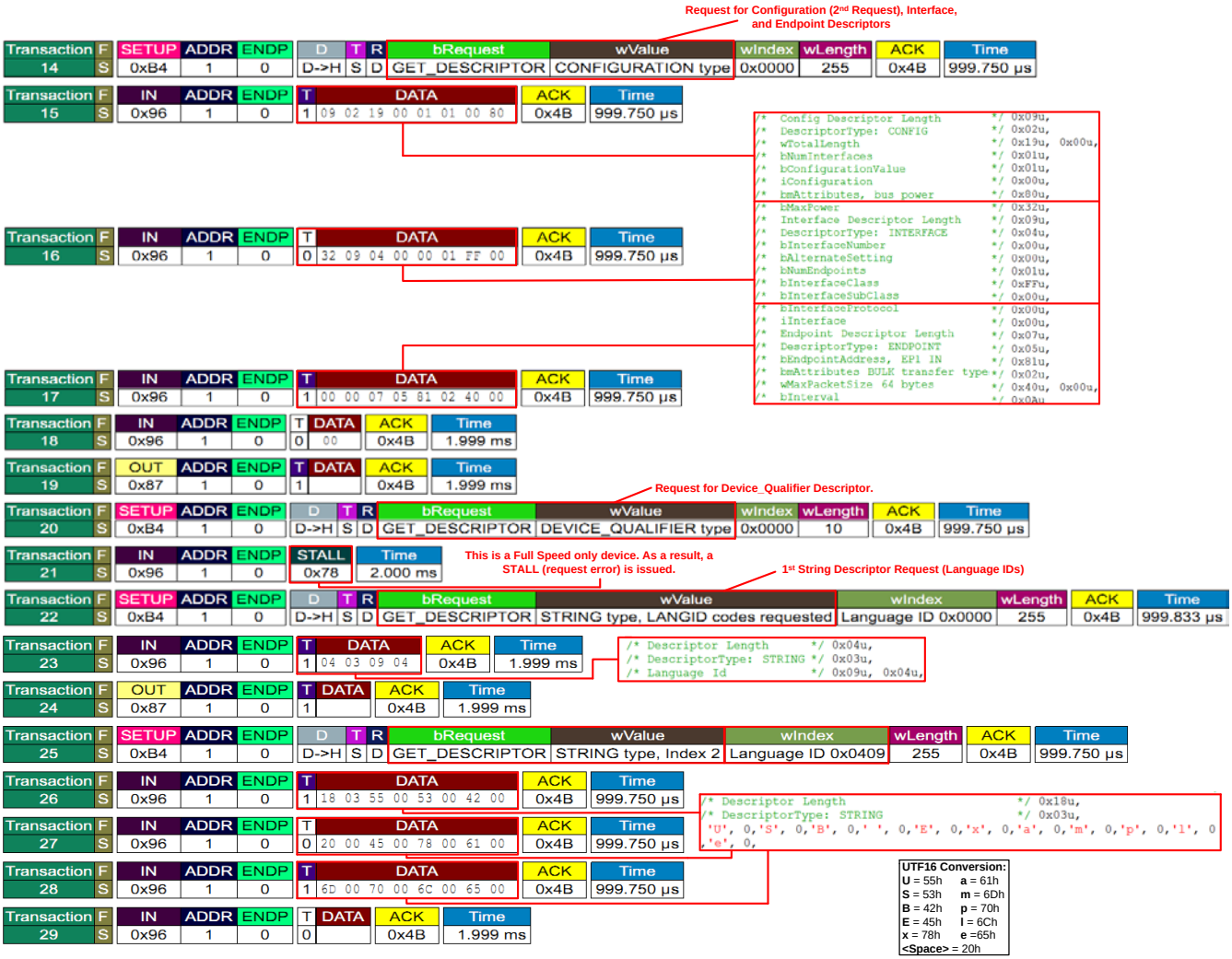
Appendix A: Example PSoC 3 Full-Speed USB Device Descriptors

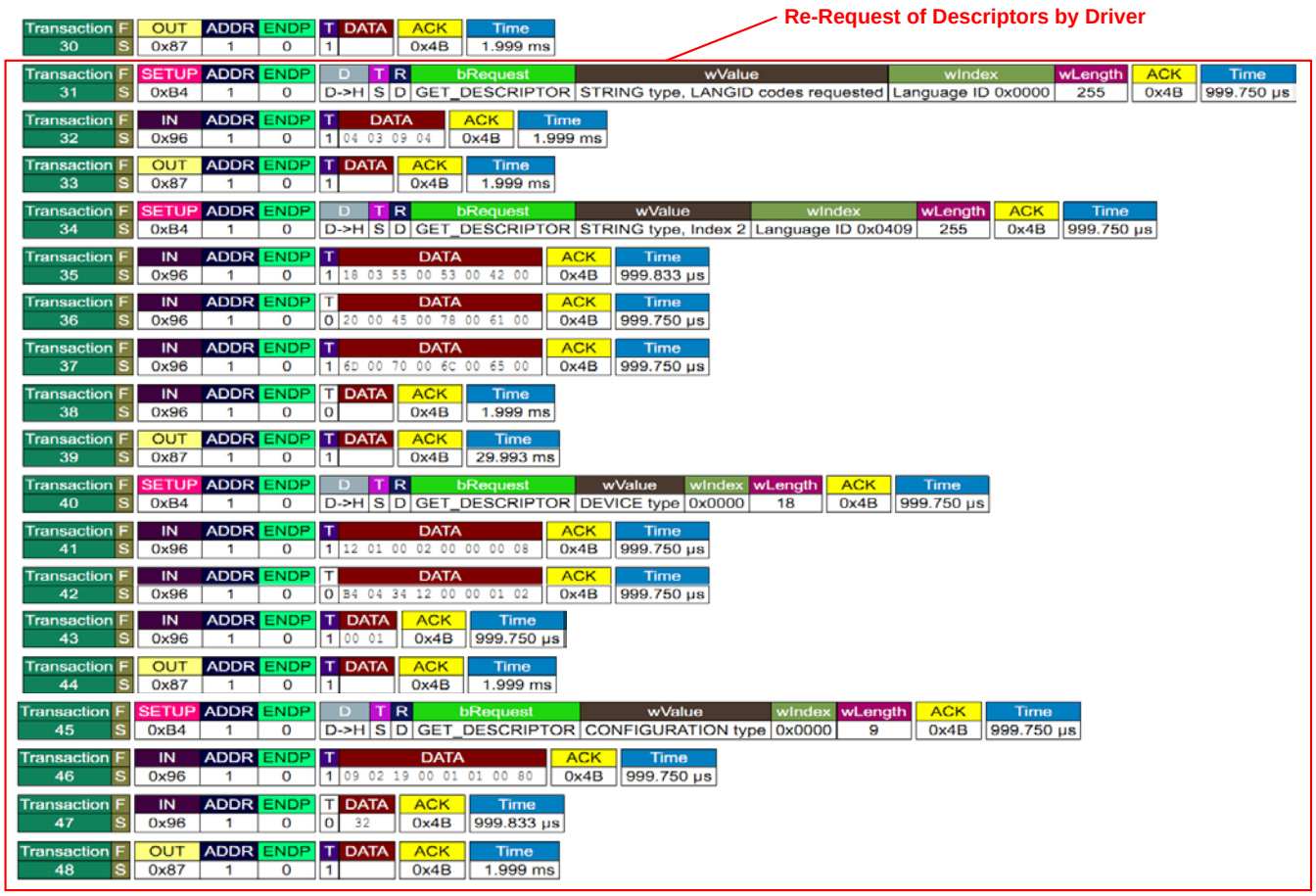
```
/*******************************************************************
 Device Descriptors
 *******************************************************************/
uint8 CYCODE USBFS_1_DEVICE0_DESCR[] = {
/* Descriptor Length                        */ 0x12u,
/* DescriptorType: DEVICE                   */ 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_DESCRIPTORS[] = {
/*****************************************************************
 Language ID Descriptor
 *****************************************************************/
/* Descriptor Length                      */ 0x04u,
 /* DescriptorType: STRING                 */ 0x03u,
 /* Language Id                            */ 0x09u, 0x04u,
/*****************************************************************
 String Descriptor: "Cypress Semiconductor"
 *****************************************************************/
/* Descriptor Length                      */ 0x2Cu,
/* DescriptorType: STRING                 */ 0x03u,
 'C', 0,'y', 0,'p', 0,'r', 0,'e', 0,'s', 0,'s', 0,' ', 0,'S', 0,'e', 0
,'m', 0,'i', 0,'c', 0,'o', 0,'n', 0,'d', 0,'u', 0,'c', 0,'t', 0,'o', 0
,'r', 0,
/*****************************************************************
 String Descriptor: "USB Example"
 *****************************************************************/
/* 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,
/*****************************************************************/
/* Marks the end of the list.             */ 0x00u};
/*****************************************************************/0,
```

Appendix B: Bus Analyzer Capture of USB Enumeration (Example)




Request for Configuration (2nd Request), Interface, and Endpoint Descriptors
Request for Device_Qualifier Descriptor.
This is a Full Speed only device. As a result, a STALL (request error) is issued.
1st String Descriptor Request (Language IDs)
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 S | OUT | 0x87 | ADDR 1 | ENDP 0 | T 1 | DATA | ACK 0x4B | Time 1.999 ms
Transaction 31 S | SETUP | 0xB4 | ADDR 1 | ENDP 0 | D D->H | T S | R D | bRequest GET_DESCRIPTOR | wValue STRING type, LANGID codes requested | wIndex Language ID 0x0000 | wLength 255 | ACK 0x4B | Time 999.750 µs
Transaction 32 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 04 03 09 04 | ACK 0x4B | Time 1.999 ms
Transaction 33 S | OUT | 0x87 | ADDR 1 | ENDP 0 | T 1 | DATA | ACK 0x4B | Time 1.999 ms
Transaction 34 S | SETUP | 0xB4 | ADDR 1 | ENDP 0 | D D->H | T S | R D | bRequest GET_DESCRIPTOR | wValue STRING type, Index 2 | wIndex Language ID 0x0409 | wLength 255 | ACK 0x4B | Time 999.750 µs
Transaction 35 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 18 03 55 00 53 00 42 00 | ACK 0x4B | Time 999.833 µs
Transaction 36 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 0 | DATA 20 00 45 00 78 00 61 00 | ACK 0x4B | Time 999.750 µs
Transaction 37 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 6D 00 70 00 6C 00 65 00 | ACK 0x4B | Time 999.750 µs
Transaction 38 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 0 | DATA | ACK 0x4B | Time 1.999 ms
Transaction 39 S | OUT | 0x87 | ADDR 1 | ENDP 0 | T 1 | DATA | ACK 0x4B | Time 29.993 ms
Transaction 40 S | SETUP | 0xB4 | ADDR 1 | ENDP 0 | D D->H | T S | R D | bRequest GET_DESCRIPTOR | wValue DEVICE type | wIndex 0x0000 | wLength 18 | ACK 0x4B | Time 999.750 µs
Transaction 41 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 12 01 00 02 00 00 00 08 | ACK 0x4B | Time 999.750 µs
Transaction 42 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 0 | DATA B4 04 34 12 00 00 01 02 | ACK 0x4B | Time 999.750 µs
Transaction 43 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 00 01 | ACK 0x4B | Time 999.750 µs
Transaction 44 S | OUT | 0x87 | ADDR 1 | ENDP 0 | T 1 | DATA | ACK 0x4B | Time 1.999 ms
Transaction 45 S | SETUP | 0xB4 | ADDR 1 | ENDP 0 | D D->H | T S | R D | bRequest GET_DESCRIPTOR | wValue CONFIGURATION type | wIndex 0x0000 | wLength 9 | ACK 0x4B | Time 999.750 µs
Transaction 46 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 1 | DATA 09 02 19 00 01 01 00 80 | ACK 0x4B | Time 999.750 µs
Transaction 47 S | IN | 0x96 | ADDR 1 | ENDP 0 | T 0 | DATA 32 | ACK 0x4B | Time 999.833 µs
Transaction 48 S | OUT | 0x87 | ADDR 1 | ENDP 0 | T 1 | DATA | ACK 0x4B | Time 1.999 ms

Re-Request of Descriptors by Driver (Continued)

Transaction	F	SETUP	ADDR	ENDP	D	T	R	bRequest	wValue	wIndex	wLength	ACK	Time
49	S	0xB4	1	0	D->H	S	D	GET_DESCRIPTOR	CONFIGURATION type	0x0000	25	0x4B	999.833 µs

Transaction	F	IN	ADDR	ENDP	T	DATA	ACK	Time
50	S	0x96	1	0	1	09 02 19 00 01 01 00 80	0x4B	999.667 µs
51	S	0x96	1	0	0	32 09 04 00 00 01 FF 00	0x4B	999.750 µs
52	S	0x96	1	0	1	00 00 07 05 81 02 40 00	0x4B	999.750 µs
53	S	0x96	1	0	0	00	0x4B	999.750 µs
54	S	0x87	1	0	1		0x4B	1.999 ms

Request for Device Status

Transaction	F	SETUP	ADDR	ENDP	D	T	R	bRequest	wValue	wIndex	wLength	ACK	Time
55	S	0xB4	1	0	D->H	S	D	GET_STATUS	0x0000	Device Status requested	2	0x4B	999.750 µs

Transaction	F	IN	ADDR	ENDP	T	DATA	ACK	Time
56	S	0x96	1	0	1	00 00	0x4B	999.750 µs

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 though 2 are reserved and set at zero

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

Configuration is Selected.
Configuration 1

Transaction	F	SETUP	ADDR	ENDP	D	T	R	bRequest	wValue	wIndex	wLength	ACK	Time
58	S	0xB4	1	0	H->D	S	D	SET_CONFIGURATION	New configuration 1	0x0000	0	0x4B	999.750 µs

Transaction	F	IN	ADDR	ENDP	T	DATA	ACK
59	S	0x96	1	0	1		0x4B

Device is Ready for Use!

Document History

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

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.

Worldwide Sales and Design Support

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

Products

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

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

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

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