About USB

This chapter presents a quick introduction to USB. The first section in this chapter introduces the basic concepts of the USB specification Revision 2.0. The second section explores the data flow model. The third section gives details about the device operation. Lastly, the fourth section describes USB device logical organization.

INTRODUCTION

The Universal Serial Bus (USB) is an industry standard maintained by the USB Implementers Forum (USB-IF) for serial bus communication. The USB specification contains all the information about the protocol such as the electrical signaling, the physical dimension of the connector, the protocol layer, and other important aspects. USB provides several benefits compared to other communication interfaces such as ease of use, low cost, low power consumption and, fast and reliable data transfer.

BUS TOPOLOGY

USB can connect a series of devices using a tiered star topology. The key elements in USB topology are the *host*, *hubs*, and *devices*, as illustrated in Figure 1-1. Each node in the illustration represents a USB hub or a USB device. At the top level of the graph is the root hub, which is part of the host. There is only one host in the system. The specification allows up to seven tiers and a maximum of five non-root hubs in any path between the host and a device. Each tier must contain at least one hub except for the last tier where only devices are present. Each USB device in the system has a unique address assigned by the host through a process called *enumeration*.

The host learns about the device capabilities during enumeration, which allows the host operating system to load a specific driver for a particular USB device. The maximum number of peripherals that can be attached to a host is 127, including the root hub.

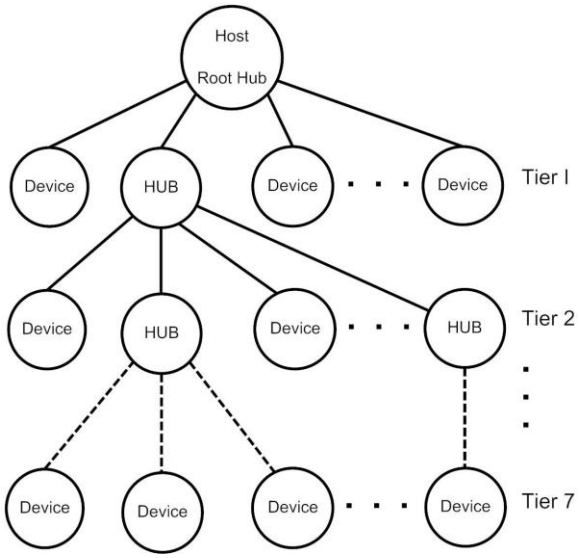


Figure 1-1 Bus topology

USB HOST

The USB host communicates with the devices using a USB host controller. The host is responsible for detecting and enumerating devices, managing bus access, performing error checking, providing and managing power, and exchanging data with the devices.

USB DEVICE

A USB device implements one or more USB *functions* where a function provides one specific capability to the system. Examples of USB functions are keyboards, webcam, speakers, or a mouse. The requirements of the USB functions are described in the USB class specification. For example, keyboards and mice are implemented using the Human Interface Device (HID) specification.

USB devices must also respond to requests from the host. For example, on power up, or when a device is connected to the host, the host queries the device capabilities during enumeration, using standard requests.

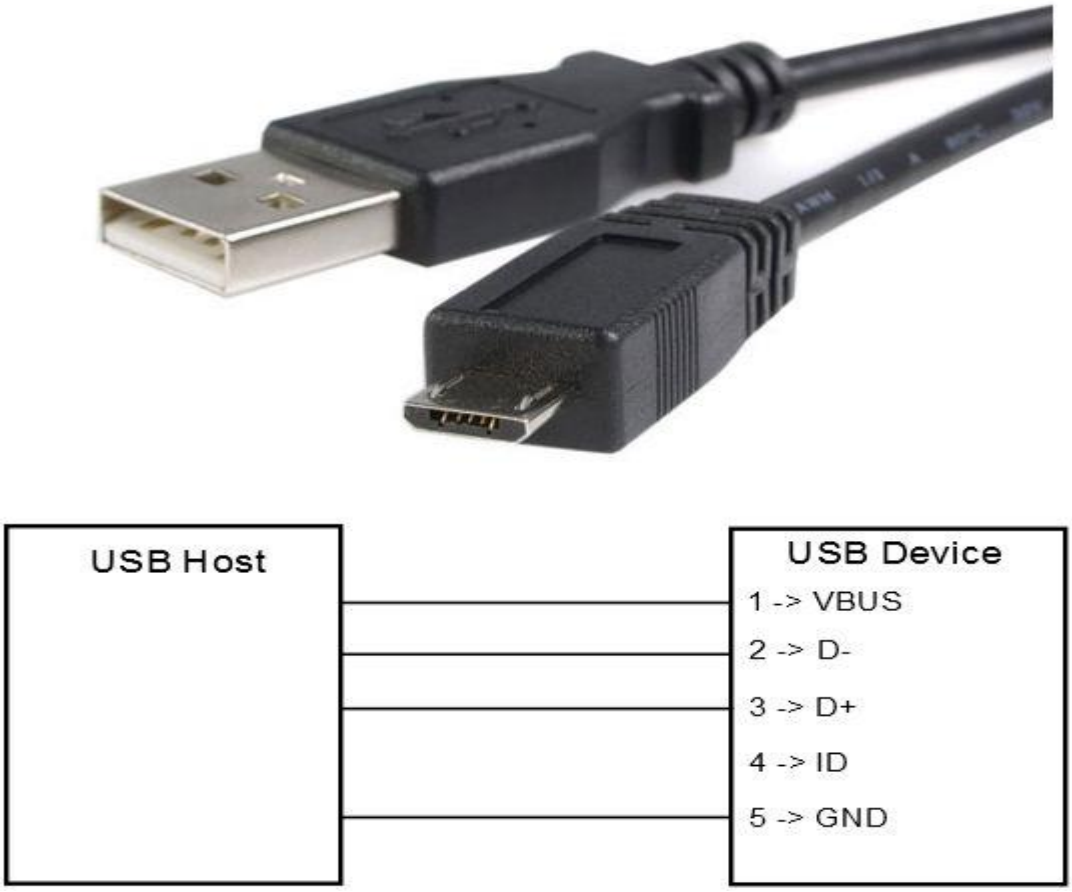


Figure 1-2 Connection in Device Mode

When a USB cable with standard-A receptacle on one end and Micro-A plug on the other end of the cable is plugged to the 'TARGET USB' connector, the ID pin will be pulled LOW. The application should detect this change and re-initialize the USB module as host. To achieve this, an interrupt needs to be configured for PC16. Inside the corresponding interrupt service routine, the application can initialize host. When PC16 goes LOW, output of an onboard voltage regulator is enabled to power the VBUS line.

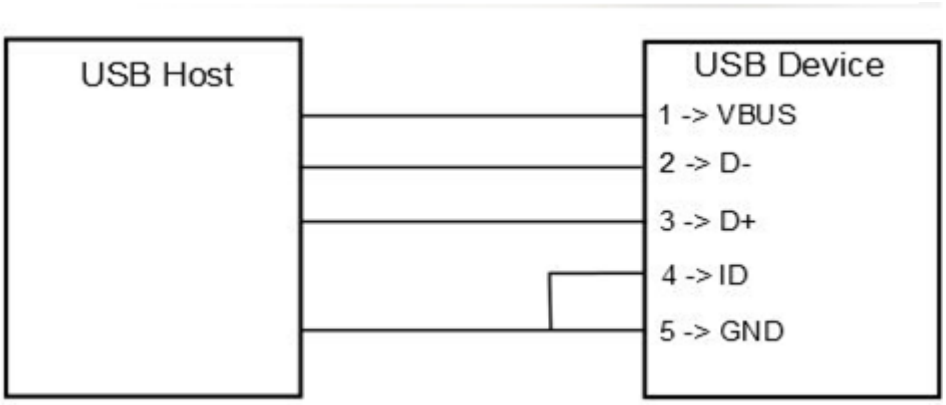


Figure 1-2 Connection in Host Mode

DATA FLOW MODEL

This section defines the elements involved in the transmission of data across USB.

ENDPOINT

Endpoints function as the point of origin or the point of reception for data. An endpoint is a logical entity identified using an endpoint address. The endpoint address of a device is fixed, and is assigned when the device is designed, as opposed to the device address, which is assigned by the host dynamically during enumeration. An endpoint address consists of an endpoint number field (0 to 15), and a direction bit that indicates if the endpoint sends data to the host (IN) or receives data from the host (OUT). The maximum number of endpoints allowed on a single device is 32.

Endpoints contain configurable characteristics that define the behavior of a USB device:

- Bus access requirements
- Bandwidth requirement
- Error handling
- Maximum packet size that the endpoint is able to send or receive
- Transfer type
- Direction in which data is sent and receive from the host

ENDPOINT ZERO REQUIREMENT

Endpoint zero (also known as Default Endpoint) is a bi-directional endpoint used by the USB host system to get information, and configure the device via standard requests. All devices must implement an endpoint zero configured for control transfers (see section “Control Transfers” on page 18 for more information).

PIPES

A USB pipe is a logical association between an endpoint and a software structure in the USB host software system. USB pipes are used to send data from the host software to the device’s endpoints. A USB pipe is associated to a unique endpoint address, type of transfer, maximum packet size, and interval for transfers.

The USB specification defines two types of pipes based on the communication mode:

- Stream Pipes: Data carried over the pipe is unstructured.

- Message Pipes: Data carried over the pipe has a defined structure.

The USB specification requires a default control pipe for each device. A default control pipe uses endpoint zero. The default control pipe is a bi-directional message pipe.

TRANSFER TYPES

The USB specification defines four transfer types that match the bandwidth and services requirements of the host and the device application using a specific pipe. Each USB transfer encompasses one or more transactions that send data to and from the endpoint. The notion of transactions is related to the maximum payload size defined by each endpoint type in that when a transfer is greater than this maximum, it will be split into one or more transactions to fulfill the action.

CONTROL TRANSFERS

Control transfers are used to configure and retrieve information about the device capabilities. They are used by the host to send standard requests during and after enumeration. Standard requests allow the host to learn about the device capabilities; for example, how many and which functions the device contains. Control transfers are also used for class-specific and vendor-specific requests.

A control transfer contains three stages: Setup, Data, and Status. These stages are detailed in Table 1-1.

Stage	**Description**
Setup	The Setup stage includes information about the request. This SETUP stage represents one transaction.
Data	The Data stage contains data associated with request. Some standard and class- specific request may not require a Data stage. This stage is an IN or OUT directional transfer and the complete Data stage represents one ore more transactions.
Status	The Status stage, representing one transaction, is used to report the success or failure of the transfer. The direction of the Status stage is opposite to the direction of the Data stage. If the control transfer has no Data stage, the Status stage always is from the device (IN).

Table 1-1 **Control Transfer Stages**

BULK TRANSFERS

Bulk transfers are intended for devices that exchange large amounts of data where the transfer can take all of the available bus bandwidth. Bulk transfers are reliable, as error detection and retransmission mechanisms are implemented in hardware to guarantee data integrity. However, bulk transfers offer no guarantee on timing. Printers and mass storage devices are examples of devices that use bulk transfers.

INTERRUPT TRANSFERS

Interrupt transfers are designed to support devices with latency constrains. Devices using interrupt transfers can schedule data at any time. Devices using interrupt transfer provides a polling interval which determines when the scheduled data is transferred on the bus. Interrupt transfers are typically used for event notifications.

ISOCHRONOUS TRANSFERS

Isochronous transfers are used by devices that require data delivery at a constant rate with a certain

degree of error-tolerance. Retransmission is not supported by isochronous transfers. Audio and video devices use isochronous transfers.

USB DATA FLOW MODEL

Table 1-2 shows a graphical representation of the data flow model.

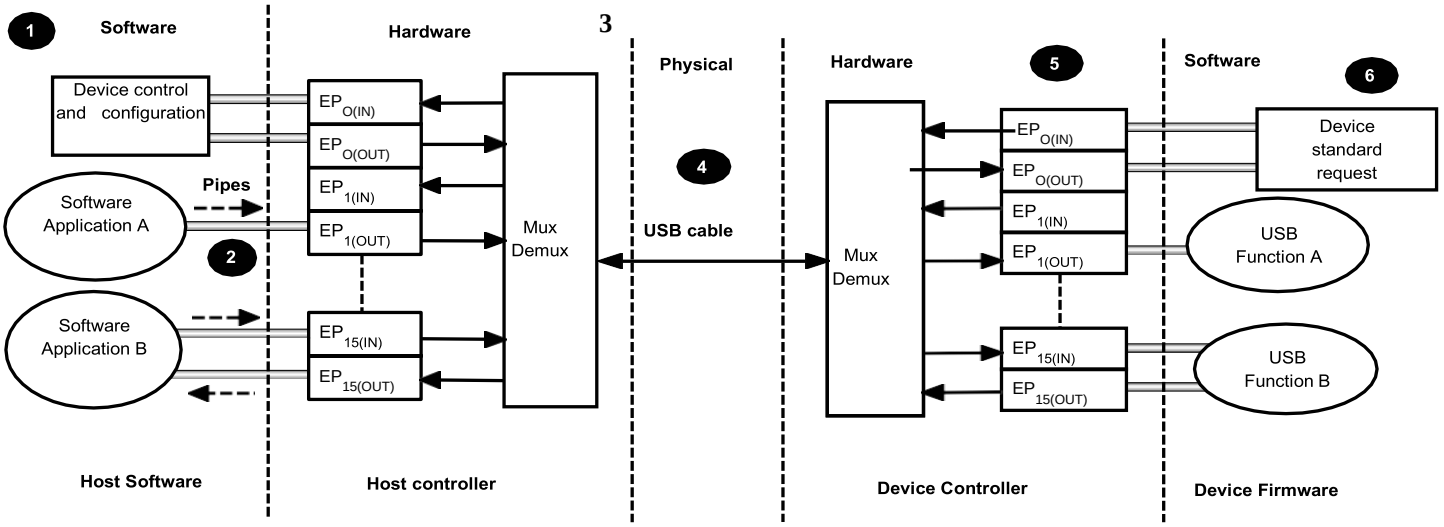


Figure 1-4 USB data flow

F1-2(1) The host software uses standard requests to query and configure the device using the default pipe. The default pipe uses endpoint zero (EP0).

F1-2(2) USB pipes allow associations between the host application and the device's endpoints. Host applications send and receive data through USB pipes.

F1-2(3) The host controller is responsible for the transmission, reception, packing and unpacking of data over the bus.

F1-2(4) Data is transmitted via the physical media.

F1-2(5) The device controller is responsible for the transmission, reception, packing and unpacking of data over the bus. The USB controller informs the USB device software layer about several events such as bus events and transfer events.

F1-2(6) The device software layer responds to the standard request, and implements one or more USB functions as specified in the USB class document.

TRANSFER COMPLETION

The notion of transfer completion is only relevant for control, bulk and interrupt transfers as isochronous transfers occur continuously and periodically by nature. In general, control, bulk and interrupt endpoints

must transmit data payload sizes that are less than or equal to the endpoint's maximum data payload size. When a transfer's data payload is greater than the maximum data payload size, the transfer is split into several transactions whose payload is maximum-sized except the last transaction which contains the remaining data. A transfer is deemed complete when:

- The endpoint transfers exactly the amount of data expected.
- The endpoint transfers a short packet that is a packet with a payload size less than the maximum.
- The endpoint transfers a zero-length packet.

PHYSICAL INTERFACE AND POWER MANAGEMENT

USB transfers data and provides power using four-wire cables. The four wires are: V_{bus}, D^+, D^- and Ground. Signaling occurs on the D^+ and D^- wires.

SPEED

The USB 2.0 specification defines three different speeds.

- Low Speed: 1.5 Mb/s
- Full Speed: 12 Mb/s
- High Speed: 480 Mb/s

POWER DISTRIBUTION

The host can supply power to USB devices that are directly connected to the host. USB devices may also have their own power supplies. USB devices that use power from the cable are called bus-powered devices. Bus-powered device can draw a maximum of 500 mA from the host. USB devices that have alternative source of power are called self- powered devices.

DEVICE STRUCTURE AND ENUMERATION

Before the host application can communicate with a device, the host needs to understand the capabilities of the device. This process takes place during device enumeration. After enumeration, the host can assign and load a specific driver to allow communication between the application and the device.

During enumeration, the host assigns an address to the device, reads descriptors from the device, and selects a configuration that specifies power and interface requirements. In order for the host learns about the device's capabilities, the device must provide information about itself in the form of descriptors.

This section describes the device logical organization from the USB host's point of view.

USB DEVICE STRUCTURE

From the host point of view, USB devices are internally organized as a collection of configurations, interfaces and endpoints.

CONFIGURATION

A USB configuration specifies the capabilities of a device. A configuration consists of a collection of USB interfaces that implement one or more USB functions. Typically only one configuration is required for a given device. However, the USB specification allows up to 255 different configurations. During enumeration, the host selects a configuration. Only one configuration can be active at a time. The device uses a *configuration descriptor* to inform the host about a specific configuration's capabilities.

INTERFACE

A USB interface or a group of interfaces provides information about a function or class implemented by the device. An interface can contain multiple mutually exclusive settings called *alternate settings*. The device uses an *interface descriptor* to inform the host about a specific interface's capabilities. Each interface descriptor contains a class, subclass, and protocol codes defined by the USB-IF, and the number of endpoints required for a particular class implementation.

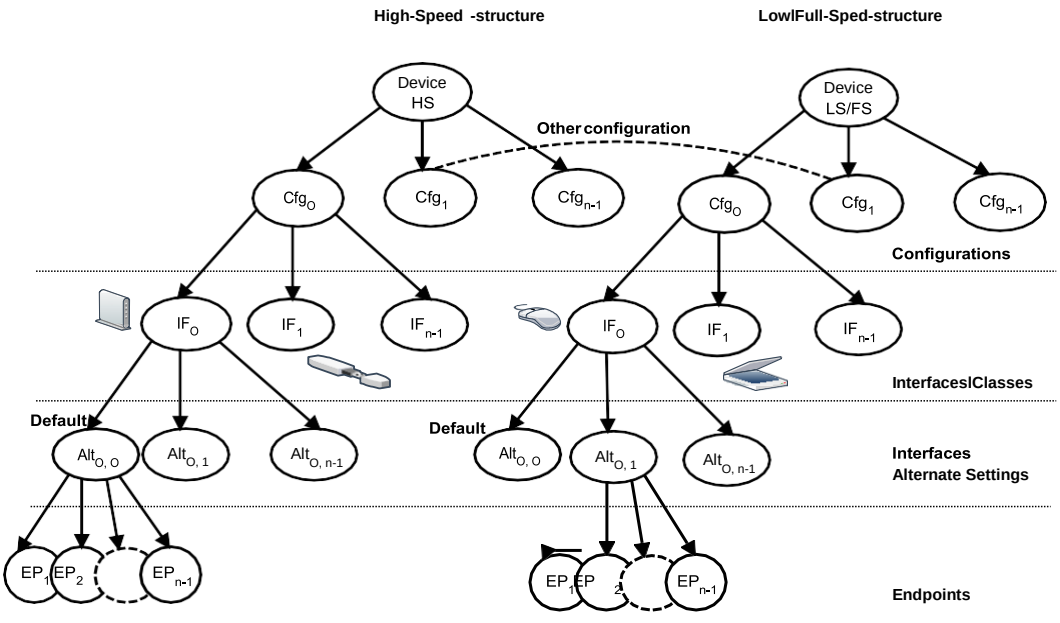
ALTERNATE SETTINGS

Alternate settings are used by the device to specify mutually exclusive settings for each interface. The default alternate settings contain the default settings of the device. The device also uses an interface descriptor to inform the host about an interface's alternate settings.

ENDPOINT

An interface requires a set of endpoints to communicate with the host. Each interface has different requirements in terms of the number of endpoints, transfer type, direction, maximum packet size, and maximum polling interval. The device sends an endpoint descriptor to notify the host about endpoint capabilities.

Figure 1-3 shows the hierarchical organization of a USB device. Configurations are grouped based on the device's speed. A high-speed device might have a particular configuration in both high-speed and low/full speed.



DEVICE STATES

The USB 2.0 specifications define six different states and are detailed in Table 1-2.

Device States	Description
Attached	The device is in the Attached state when it is connected to the host or a hub port. The hub must be connected to the host or to another hub.
Powered	A device is considered in the Powered state when it starts consuming power from the bus. Only bus-powered devices use power from the host. Self-powered devices are in the Powered state after port attachment.
Default	After the device has been powered, it should not respond to any request or transactions until it receives a reset signal from the host. The device enters in the Default state when it receives a reset signal from the host. In the Default state, the device responds to standard requests at the default address 0.
Address	During enumeration, the host assigns a unique address to the device. When this occurs, the device moves from the Default state to the Address state.
Configured	After the host assigns an address to the device, the host must select a configuration. After the host selects a configuration, the device enters the Configured state. In this state, the device is ready to communicate with the host applications.
Suspended	The device enters in Suspended state when no traffic has been seen in the bus for a specific period of time. The device retains the address assigned by the host in the Suspended state. The device returns to the previous state after traffic is present in the bus.

ENUMERATION

Enumeration is the process where the host configures the device and learns about the device's capabilities. The host starts enumeration after the device is attached to one of the root or external hub ports. The host learns about the device's manufacturer, vendor/product IDs and release versions by sending a *Get Descriptor* request to obtain the device descriptor and the maximum packet size of the default pipe (control endpoint 0). Once that is done, the host assigns a unique address to the device which will tell the device to only answer requests at this unique address. Next, the host gets the capabilities of the device by a series of *Get Descriptor* requests. The host iterates through all the available configurations to retrieve information about number of interfaces in each configuration, interfaces classes, and endpoint parameters for each interface and will lastly finish the enumeration process by selecting the most suitable configuration.

USB Device Stack

Introduction

This document introduces the USB device stack. This stack is included in the Advanced Software Framework (ASF), and aims to provide the customer with the quickest and easiest way to build a USB application. A full description of this stack is available in this document. Only basic knowledge of USB is required to use this stack.

Abbreviations:

- **APP: User Application**
- **ASF: Advanced Software Framework**
- **CBW: Command Block Wrapper (from Mass Storage Class)**
- **CDC: Communication Device Class**
- **CSW: Command Status Wrapper (from Mass Storage Class)**
- **DP or D+ Data Plus Differential Line**
- **DM or D- Data Minus Differential Line**
- **FS: USB Full Speed**
- **HID: Human Interface Device**
- **HS: USB High Speed**
- **UDC: USB Device Controller**
- **UDD: USB Device Descriptor**
- **UDI: USB Device Interface**
- **USB: Universal Serial Bus**
- **MSC: Mass Storage Class**
- **PHDC: Peripheral Health Device Class**
- **sleepmgr: Sleep Management Service from ASF**
- **ZLP: Zero Length Packet**

USB Device Application Notes:

Several USB device examples are provided by Microchip. Some of these examples are covered by their own application note.

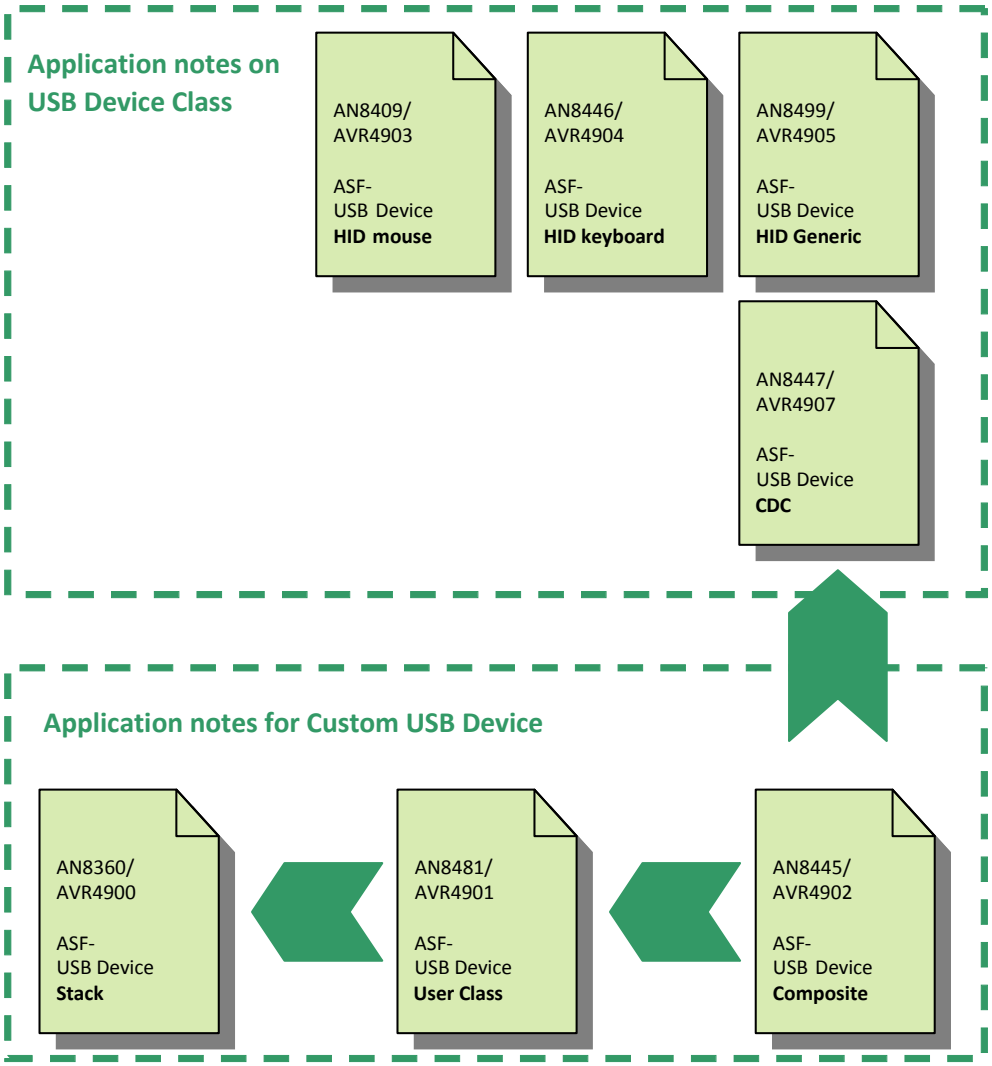
Basic USB knowledge is necessary to understand the USB device class application notes (Classes: HID, and CDC).

To create a USB device with one of the ASF provided classes, refer directly to the related application note for this USB class.

The new class and composite USB device application notes are designed for advanced USB developers.

Other examples are also available in Atmel Studio. To list these, select "New Example Project..." from the start screen or the File menu (File → New → Example Project...) in Atmel Studio 7. The list of examples can be reduced to list only USB by either searching for USB in the search field or select USB from the technology tab.

Figure 2-1. USB Device Application Notes



Organization:

Overview

The USB device stack is divided into three parts:

- USB Device Controller (UDC) Provides USB Chapter 9 Compliance
- USB Device Interface (UDI) Provides USB Class Compliance
- USB Device Driver (UDD) Provides the USB Interface for Each AVR Product

USB Device Stack

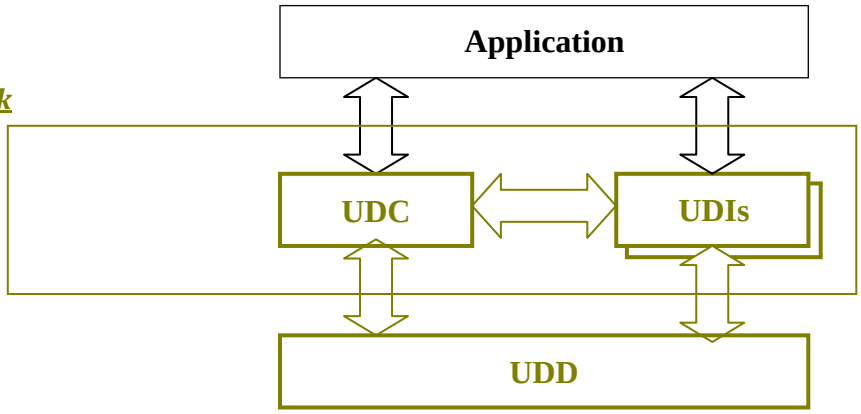
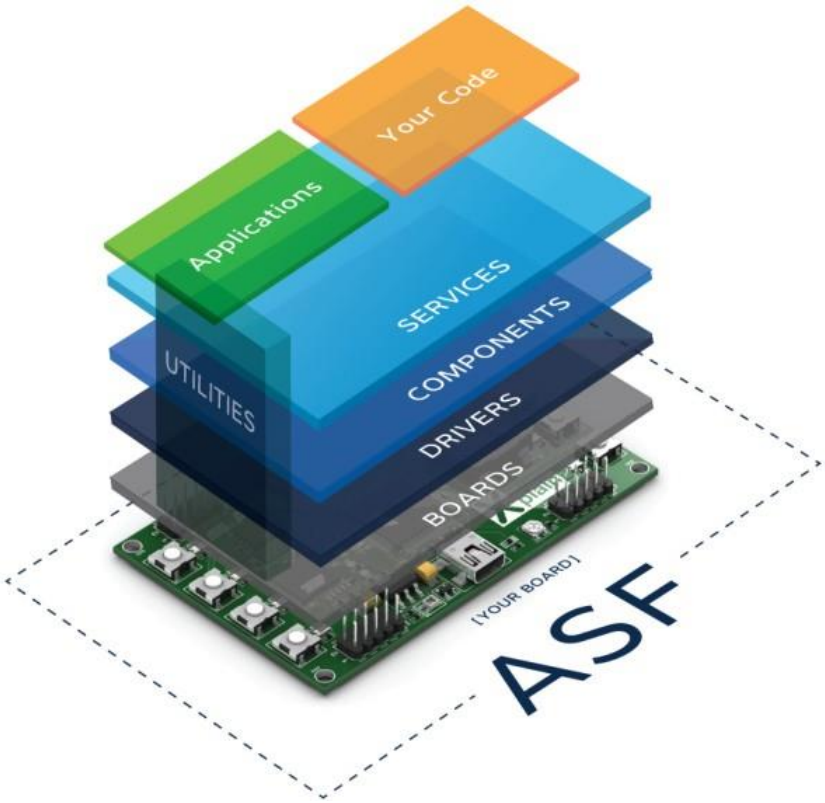


Figure 3-1. USB Device Stack Architecture

NOTE: For USB Device Stack Details refer the given pdf document.
Atmel-42336-ASF-USB-Stack-Manual.pdf.

USB Device Stack:
The API interfaces to the USB Stack for applications, included in ASF as middleware service.



SD_MMC Driver:

Add the SD_MMC Driver from the ASF wizards. To know the overall description of these protocol refer the below mentioned document.

Mmc_spec.pdf.

Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. It is compliant with the American FIPS (Federal Information Processing Standard) Publication 197 specification.

The AES supports all five confidentiality modes of operation for symmetrical key block cipher algorithms (as specified in the NIST Special Publication 800-38A Recommendation):

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Output Feedback (OFB)
- Cipher Feedback (CFB)
- Counter (CTR)

Data transfers both to and from the AES module can occur using the peripheral DMA controller channels, thus minimizing processor intervention for large data buffer transfers.

As soon as the initialization vector, the input data and the key are configured, the encryption/decryption process may be started. Once the process has completed the encrypted/decrypted data can be read out via registers.

Encryption and Decryption

The AES is capable of using cryptographic keys of 128/192/256 bits to encrypt and decrypt data in blocks of 128 bits. In Cipher Feedback Mode (CFB), five data sizes are possible (8, 16, 32, 64, or 128 bits). The input to the encryption processes of the CBC, CFB, and OFB modes includes, in addition to the plaintext, a 128-bit data block called the Initialization Vector (IV). The Initialization Vector is used in the initial step in the encryption of a message and in the corresponding decryption of the message.

There are three encryption/decryption start modes:

- Manual Mode: Start encryption/decryption manually
- Auto Start Mode: Once the correct number of input data registers is written, processing is automatically started, DMA operation uses this mode
- Last Output Data Mode (LOD): This mode is used to generate message authentication code (MAC) on data in CCM mode of operation

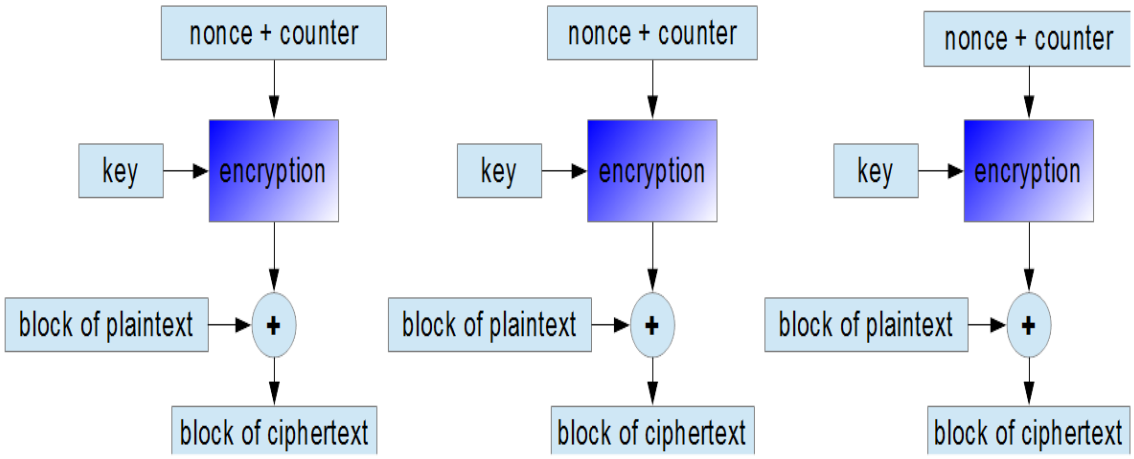
Encryption and Decryption Implementation in SCSI layer:

The encryption and decryption was implemented at SCSI layer. In SCSI Layer we found the read/write buffer, that buffer was encrypted at write time and decrypted at read time. Those encrypted and decrypted functions will elaborate below.

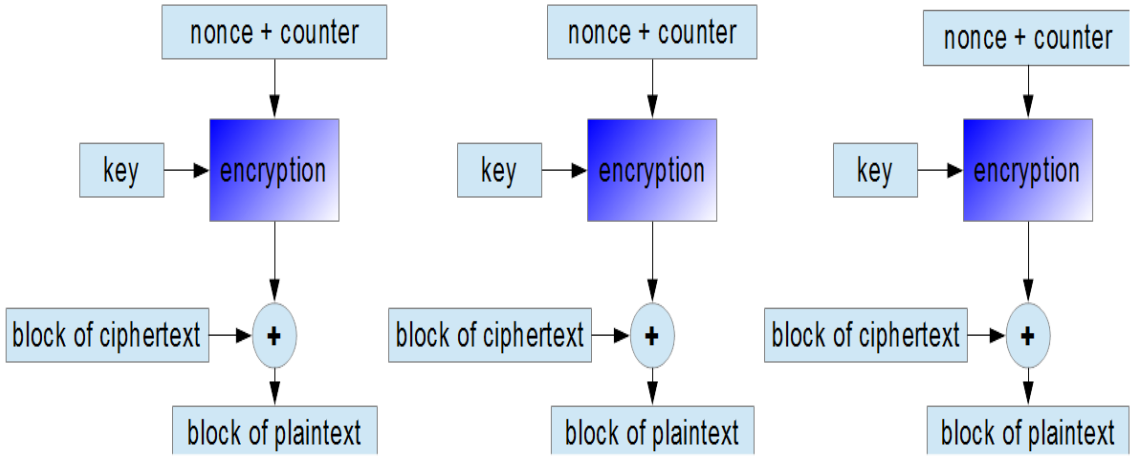
In this process we have chosen the counter (CTR) mode symmetrical key block cipher algorithm because of its advantages.

CTR (Counter) Mode

Using the CTR mode makes block cipher way of working similar to a stream cipher. As in the OFB mode, key (256-bits) stream bits are created regardless of content of encrypting data blocks. In this mode, subsequent values of an increasing counter are added to a *nonce* value (the nonce means a number that is unique: *number used once*) and the results are encrypted as usual. The nonce plays the same role as initialization vectors in the previous modes.



Encryption in the CTR mode



Decryption in the CTR mode

It is one of the most popular block ciphers modes of operation. Both encryption and decryption can be performed using many threads at the same time.

If one bit of a plaintext or cipher text message is damaged, only one corresponding output bit is damaged as well. Thus, it is possible to use various correction algorithms to restore the previous value of damaged parts of received messages.

The CTR mode is also known as the SIC mode (Segment Integer Counter).

Security of the CTR mode

As in the case of the CBC mode, one should change the secret key after using it for encrypting a number of sent messages. It can be proved that the CTR mode generally provides quite good security and that the secret key needs to be changed less often than in the CBC mode.

Functions implemented at SCSI layer level:

In main.c file the below given function will initializes the AES encryption and decryption related functionalities.

cipher_init (); //initializes the AES module.

At SCSI layer level inside the sd_mmc_mem.c file we have the read/write functions. We were encrypting the data at the writing time and decrypting at the time of reading time. Those functions given as follows;

ENCRYPTION:

Ctrl_status sd_mmc_usb_write_10 (*uint8_t* slot, *uint32_t* addr, *uint16_t* nb_sector)

Inside the above given function we are encrypting data while writing data from RAM to MCI. The function is given below which is going to performs the encryption.

encrypt_write_data (((nb_step % 2) == 0) ? sector_buf_0 : sector_buf_1, SD_MMC_BLOCK_SIZE); //performs the encryption

DECRYPTION:

Ctrl_status sd_mmc_usb_read_10 (*uint8_t* slot, *uint32_t* addr, *uint16_t* nb_sector)

Inside the above given function we are decrypting data while reading data from MCI to RAM. The function is given below which is going to performs the decryption.

decrypt_read_data (((nb_step % 2) == 0) ? sector_buf_1 : sector_buf_0, SD_MMC_BLOCK_SIZE); //performs the decryption.