The Universal Serial Bus from Abstraction to Implementation

by Mohammed Fennich
Intel Corporation.

When a Universal Serial Bus (USB) device is attached to or detached from a USB host, a set of predetermined events take place. The USB Specification, Rev. 1.0, explains these events and dictates the requirements for both the device and the USB host. The objective of this paper is to shed some light on the USB Specification section dealing with these events, bringing them from an abstract level to a more tangible and practical implementation level. In addition to a USB overview, this article provides clarifications on USB transactions, attach/detach operations, and device enumeration processes. The Intel 8x930 USB microcontroller is used to illustrate the implementation and the programming models involved.

Introduction

The Universal Serial Bus technology comes as a response to the increasing demands of our computerized and automated life, and the
need for flexible easy-to-use products. Computer-telephone integration and video conferencing through digital cameras and computers have become a necessity in the work place and are starting to make their way into homes. These two interrelated communication functions, along with a number of standard PC peripherals, such as keyboards, joysticks and mice, will benefit from USB usage.

USB is a communication protocol that supports serial data transfers between a USB host computer and USB-capable peripherals. The host serves as the master of the bus. Data transfer is serial with two modes of signaling: full-speed mode with a signaling rate of 12 Mbs, and low-speed mode with a signaling rate of 1.5 Mbs. The peripherals perform as slaves connected either directly to the host or through hubs, in a tiered star topology, with a hub at the center of each star as shown in Figure 1.

USB addresses issues such as connectivity to the PC, ease-of-use, flexibility, and low-cost implementations. This is in addition to the "plug and play" feature which is one of the main reasons behind the inception of USB. With USB, users can connect or disconnect a peripheral without having to reconfigure or alter the setup, or be concerned with what

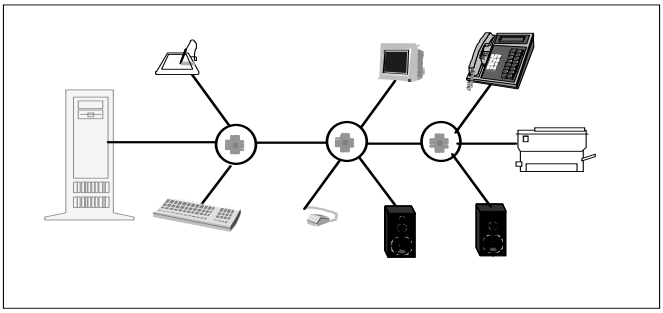

Figure-1 USB system example

plugs to use for their peripherals, because the connection type is standardized. Also, users don't have to install an add-in card for each peripheral they want to connect to their PC.

Most or all of the functionality of the add-in cards has been implemented in software. In fact, the USB host PC is equipped with a sophisticated software structure that automatically and transparently loads drivers corresponding to the plugged devices. This makes it possible for users to plug their peripherals to the PC without having to power down or reboot the system.

The USB Host

USB hosts are PCs built around USB-capable motherboards and are equipped with USB software structure. On the motherboard, the USB Serial Interface Engine module is incorporated as part of the USB host controller. Intel implements the USB functionality as part of the PCI chip set. The software structure is divided into three sections:

- The host controller driver that links the different host controller implementations with the rest of the USB software structure.
- USB system software that acts as a layer between the host controller and the client software.
- Client software that allows the client to communicate with their attached USB device through a set of abstractions within the USB software layers.

There is only one USB host on any USB system. Using USB hardware and software structures, the host acts as the master of the bus. It acknowledges the attachment and removal of devices and initiates the enumeration process and all subsequent USB transactions along the bus, and is responsible for collecting status and activity statistics and controlling the electrical interface between the host controller and the USB device.

The USB Cable

The USB cable consists of four wires. Two wires (D^+ and D^-), are used for actual information transfer using differential signaling. The two other wires are used for power--Vbus and ground--Gnd. Low-cost, unshielded cables are usually used for devices using the 1.5Mbs low-speed USB rate. Peripherals using the 12Mbs rate require shielded cable.

A USB cable should have an A type plug on one end for upstream connections (towards the USB host), and a B type plug on the other end for downstream connections (towards the peripheral). In order to eliminate customer confusion and illegal connections, type A and B connectors are not interchangeable. The USB specification provides the mechanical characteristics for both connectors.

All downstream ports have pull down resistors on both the D^+ and D^- positions. Full-speed devices should have a pull-up resistor for the D^+ wire and low-speed devices should have the pull-up resistor for the D^- wire. These resistors cause different voltage levels to occur between the D^+ and D^- lines. This implementation allows the host to sense attach and detach operations and determine what speed the connected peripheral will support.

The USB Peripherals

USB peripherals (devices) act as slaves on the bus and are of two types: hubs and functions. A hub typically consists of a hub controller and a repeater. Each hub converts a single attachment point into multiple attachment points. For example, a hub can have one upstream connector and 4 downstream ports to which other hubs and/or functions can be attached. Functions are PC peripherals like mice, keyboards, joysticks, cameras, etc. For practical reasons, hubs can be incorporated inside keyboards and monitors to allow the user to connect a USB peripheral. All USB devices, hubs, and functions must be USB specification compliant in order to be recognized by the host.

In general, USB devices consist of three components:

- A Serial Interface Engine (SIE), implemented in silicon. It is responsible for the transmission and reception of USB structured data.

- A hardware and firmware combination responsible for data transfer between the SIE and the device endpoints and their corresponding pipes.

- The third element corresponds to the actual capability or functionality that the device brings to the system (e.g., keyboard functionality)

An endpoint is the ultimate source or sink of data on the USB peripheral. At the implementation level, it can be thought of as a set of memory locations to which data can be written, or read from depending on the direction of the data flow.

A pipe is a software association to the endpoint on the USB host. At the implementation level, pipes can be thought of as software channels using function calls within the USB system software in order to send information to their associated endpoints.

Any simultaneously attached peripherals will share the USB bandwidth through a host-scheduled token-based protocol. When the host broadcasts a token over the bus, a device that detects a match with its address (in the token) responds to the host.

The USB Protocol

The USB host keeps the bus constantly active by sending a start of frame (SOF) packet every 1ms. The available bus bandwidth is shared between simultaneously connected devices within the 1ms time interval. In general, USB transactions consists of up to three packets: a token

packet, a Data Packet and a handshake packet. Figure 2 shows the different packet formats. Each packet has a packet ID (PID) that specifies its type. A transaction starts when the host controller sends a token packet with a device address, and endpoint number, the direction of data transfer, and the type of the pipe supported. The addressed device selects itself by decoding its address from the token. If the direction field in the token indicates that the host is asking for data, then the device responds with a data packet, otherwise, the host follows up with the data packet. In general, after the data is received, the destination (host or device) sends a handshake packet . A handshake packet can either be an ACK, NAK, or a STALL. (Refer to the USB Specification, Rev. 1, for more details.)

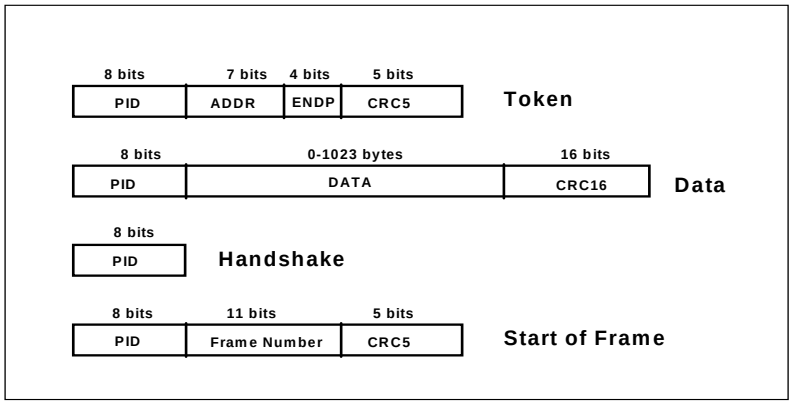

Figure-2 Packet Formats

The USB Data Transfer Categories

USB data exchange takes place between the host software and a particular device endpoint. Information flows either in a bidirectional or unidirectional manner. The host initiates the data transfer and exchanges data with each endpoint independently from the others.

Four types of transfers are supported by USB: control, interrupt, isochronous, and bulk.

*1. Control transfer*s are supported by a bidirectional communication flow. These transfers are mainly used to interrogate and configure the device at connection. However, they can be used in implementation-specific ways by client software drivers. At the protocol level, control transfers consist of a setup token stage and a status (handshake feedback) stage with a possible data stage in between. All peripheral devices have to support control transfers. The host assumes that endpoint 0 is used in association with the control pipe.

2. *Interrupt transfers* are supported by a unidirectional communication flow. These type of transfers are used for peripheral devices like mice, keyboards and joysticks. Because the USB host is the master in all USB systems, it cannot be interrupted. This implies that the actual interrupts caused by the peripherals are polled. In other words,

the peripherals don't really interrupt the host. This is usually transparent to the end-user but developers have to take it into account.

At the USB protocol level, interrupt transfers start with an IN token initiated by the host. The peripheral responds with a NAK if there is no new interrupt information to return. If a new interrupt is pending the function returns the information as a data packet. The host responds with an ACK if the data was received with no errors and does not respond if the data was erroneous . If the interrupt endpoint is stalled, it returns a STALL indicating that host software intervention is required.

3. *Isochronous transfers* are supported by a unidirectional communication flow. The direction of the flow can either be from the endpoint to the host, or from the host to the endpoint. This implies that two endpoints are required on peripherals that need to use both directions, or two pipes have to be associated with the same endpoint.

At the protocol level, isochronous transfers start with an IN or an OUT token, issued by the host, depending on the direction of communication (in the pipe) used and its corresponding endpoint. For example, in the case of an IN token, the function returns data. In the case of an OUT token, the host follows up with data. For isochronous transfers there are no handshakes or retries. This implies that inaccurate data will be lost. This type of implementation makes sense when you consider a phone or speakers as the peripherals using isochronous transfers.

4. *Bulk transfers* are supported by a unidirectional communication flow. The direction of the flow can either be from the endpoint to the host or from the host to the endpoint. This implies that two endpoints, or two pipes used with the same endpoint, are required on peripherals that need to use both directions.

At the protocol level, bulk transfers consist of three phases: token, data, and handshake. In the case of a stall condition there is no data phase and the transaction is reduced to a token and a handshake.

Except for control transfers, data is usually transported through a pipe between a memory buffer associated with the client software on the USB host and an endpoint on the USB device. Each endpoint has specific capabilities that are chosen by the device designer. The choice of these characteristics establishes a pipe associated with the particular endpoint. However, the same endpoint can be used for two pipes.

Attach Operation

Immediately after a device is attached to the USB host, the host controller is alerted of the attachment(s) by way of the voltage change on the D^+ and D^- lines. The host considers this a port status change. The device is then considered in the attached state. If the host is in suspend mode, the attachment of the device will wake it up.

Once the host detects that a device is connected, it determines the exact port it is connected to and enables it. The host then issues a reset signal to the device which lasts for at least 10 ms. After the 10ms period elapses, the port is considered enabled.

Devices that are performing a function at the time of attachment cannot sustain a reset, however, the USB specification requires it. In order to find a solution to this dilemma, the Intel 8x930 USB microcontroller implements an optional split reset feature. This feature allows the microcontroller side of the chip to continue executing code allowing the device to function while the

USB side of the chip detects the USB reset and responds to it.

After the reset sequence, the host can provide the device, hub, or function, with up to 100mA of power through the Vbus/Gnd lines. At this point, the device is considered to be in the powered state. It should be noted that self-powered devices will not require the 100mA power from the bus, even during the attachment sequence. In order to limit power consumption during connection time, the core of the 8x930 USB microcontroller starts at a low frequency rate of 3 MHz instead of a rate of 12 MHz. The developer is given the option to switch the frequency of the core at any time without compromising the USB rate.

The device sequentially changes states from the disconnected to attached, fully reset, and powered state, and becomes ready to respond to commands from the host. The host will use the default address (address 0) to communicate with the device and start the enumeration process.

Enumeration Process

The bus enumeration process consists of an interrogation sequence through which the USB host acquires information from the connected device, gives it a unique address, and assigns it a configuration value . The process takes four steps:

Step 1: The host issues a Get_Descriptor command to the device through the default address using the control pipe. The device then provides information about itself, such as device class, vendor id, maximum packet size for endpoint-0, etc.

Step 2: The host sends a unique address in a data packet to the device using the Set_Address command. The device, under the control of the 8x930 firmware, gets the address through endpoint 0 and stores it in a special function register.

Step-3: The hosts requests and reads the device configuration descriptor using the Get_Configuration command. The device responds with information about the number of interfaces and endpoints, endpoint transfer type, packet size and direction, maximum power requirements, power source, etc.

Step-4: The last step of the enumeration process is handled using the Set_Configuration command through which the host assigns a configuration value to the device.

After the enumeration process is complete, the device is configured and ready for USB data transmit and receive transactions. When the host enumerates the device the information gathered is passed on to the host system software. The system software either recognizes the device and loads the corresponding device driver, or prompts the user to provide a device driver for the application. After the device driver is loaded the device can start performing its intended function (e.g., mouse) until it is detached from the host.

Detach Operation

Immediately after a device is detached from a USB host port, specific voltage changes on the D^+ and D^- lines alert the host of the action. The host then disables the port through the USB host controller. The USB system software (part of the operating system) acknowledges the removal of the device and frees up any host resources the device was using. For example, the host will reclaim the bus bandwidth the device was using. If the device is bus-powered, the host reclaims the power and makes it available to other devices.

The detached device will stop providing any capability to the USB host but will continue providing its stand-alone function if applicable. For example, when a USB telephone is disconnected from the host, it will continue to provide phone services. The USB specification does not provide any conditions or requirements for detached devices, however, USB devices should be designed with dynamic connect/disconnect operations in mind.

At the firmware level, this is done by implementing a wait-loop within the USB code at which the USB controller (8x930) pauses if not executing any function-related code. The developer can pause the controller at the wait-loop, after all USB registers and endpoints are initialized, or have the initialization code ready to execute immediately after the reset signal is sent by the USB host.

If the disconnected device is a hub, then all the devices attached to the hub become detached, and the host reclaims all the resources these devices were using.

Programming model using the 8x930

The 8x930 USB microcontroller consists of an eight bit microcontroller core with on-chip memory and peripherals plus an on-chip USB module. The 8x930 supports all four types of USB data transfers: control, isochronous, interrupt, and bulk. The user can select the number of endpoint pairs and whether USB reset is separate from chip reset. Data transfers with the host are made to or from endpoint pairs on the USB module. Each endpoint pair has a transmit FIFO and receive FIFO data buffer. Depending on the application and the data transfer type, the user can choose from a set of different FIFO sizes supported.

Transmit FIFOs are written by the CPU, then read by the function interface unit for transmission. Receive FIFOs are written by the function interface unit following reception, then read by the CPU.

The hub version of the 8x930 provides a USB interface for a PC peripheral and provides USB hub capabilities, permitting the connection of additional PC peripherals or hubs. It provides four external downstream ports and one internal downstream port.

Operation of the USB module is controlled through the use of special function registers (SFRs). There are SFRs associated with the function interface and others associated with the hub operations. The developer has accessibility to these SFRs and is responsible for configuring the device functionality by putting the right values (bits and bytes) in the relevant SFRs. This is done through initialization routines the device needs to execute at start up.

Because the 8x930 incorporates a USB module in silicon, USB device designers don't have to worry about developing firmware to emulate the USB protocol. Developers only have to initialize the USB module to the specific configuration they need the peripheral to function at.

In addition to the initialization routines, the USB peripheral programming model for the 8x930 can be divided into three sections: enumeration code, transmit and receive routines, and application code. Designers are responsible for developing enumeration code so that the device can respond to the USB host enumeration commands (Get_Descriptor, Set_Address, Get_Configuration, and Set_Configuration).

The transmit operation requires three steps. The firmware is responsible for pre-transmit data preparation and post transmission data management. The actual data transmission is

done by the function interface hardware of the USB module. The receive operation takes two steps. The firmware is responsible for post data receive management. The data is received through the function interface hardware of the USB module.

The last part of the firmware on the peripheral corresponds to the capability that the peripheral adds to the USB host. For instance, if the peripheral is a keyboard, then the designer is responsible for developing key-scan routines and other keyboard management functions**.**