Freescale Semiconductor
Application Note

Document Number: AN3492
Rev. 0, 08/2007

USB and Using the CMX USB Stack

by: Eric Gregori,
Product Specilaist—Embedded Firmware

1 Overview of USB

Universal Serial Bus (USB) is not a true bus, it is a point-to-point star topology. There is a single host that initiates all data transfers (USB is a polled communication system). Up to 127 devices can be managed by a single host, with hubs providing the connections from the host to the devices. Electrically, USB supports point to point connections with a max length of 5 meters, and a maximum of 5 hubs between the host and end device.

Low-level USB communications are implemented using a protocol. The protocol is slightly different depending on the type of transfer. The USB specification describes this protocol using the terms host and function. The host is the master of the bus and starts all data transfers. The function (also called the device) is used to execute the requests of the host.

The upper levels of USB communications are referred to as classes. A class is a group of variables and methods. A method can also be called a function because it performs

Contents

freescale™ semiconductor

an action. The key to understanding USB communication is understanding the difference between a protocol and a class. USB is not a layered communication protocol such as TCP/IP. It is a single layer communication protocol, with various application specific classes defining the variables transferred between hosts and devices, and the requests from the host to execute methods defined by the class.

There are various classes defined by the USB-IF: HID, CDC, mass-storage, audio, and video. Each of these classes have unique variables (data) that are exchanged between the host and device using set data structures. Each class also has methods that are actually implemented as function in the device that the host requests to execute.

The USB Implementers Forum manages the USB specification, which can be found at www.usb.org

The USB-IF also distributes vendor IDs. Vendor IDs provide a unique mechanism to identify a product. The vendor ID is sent to the host when the device is connected. Vendor IDs can be purchased from the USB-IF for $2000.00 to $4000.00.

2 USB Communications

USB is a host-controlled communication system. All transactions are initiated by the host. Transactions consist of three packets: a token packet (always sent by the host), a data packet (can be sent by host or device), and a handshake packet (sent by either the host or device). Because only the host can send a token packet, USB is essentially a polled communications protocol. The device cannot send data to the host unless the host initiates the transfer.

Packets are a block of information with a defined data structure. The packet is the lowest level of the USB transfer hierarchy describing the physical layer of the interface.

Only the host sends the token packet, starting every transfer.

2.1 Packet Structures

Token Packet
Sent from HOST Only

Field	PID	Address	Endpoint	CRC
Bits	8	7	4	5

SOF Packet
Sent from HOST

Field	PID	Frame Number	CRC
Bits	8	11	5

Data Packet
Sent from HOST or DEVICE

Field	PID	Data	CRC
Bits	8	0–1023	16

Handshake Packet
Sent from HOST or DEVICE

Field	PID
Bits	8

Figure 1. Packet Structures

2.2 Packet Identifiers (PID)

The packet identifier (PID) determines the packet type and the direction of data transfer. The PID is only 4 bits, but the packet contains both the PID and the compliment of the PID. This is a form of checksum. The CRC is calculated on all but the PID field.

(LSb) (MSb)

PID_0	PID_1	PID_2	PID_3	$\overline{PID}_0$	$\overline{PID}_1$	$\overline{PID}_2$	$\overline{PID}_3$

Figure 2. PID Register from Universal Serial Bus Specification, Rev. 2.0

Table 1. PID Description from Universal Serial Bus Specification, Rev. 2.0

PID Type	PID Name	PID<3:0>	Description
Token	OUT	0001B	Address + endpoint number in host-to-function transaction
	IN	1001B	Address + endpoint number in function-to-host transaction
	SOF	0101B	Start-of-frame marker and frame number
	SETUP	1101B	Address + endpoint number in host-to-function transaction for SETUP to a control pipe
Data	DATA0	0011B	Data packet PID even
	DATA1	1011B	Data packet PID odd
	DATA2	0111B	Data packet PID high speed, high bandwidth isochronous transaction in a microframe (see 5.9.3 for more information)
	MDATA	1111B	Data packet PID high speed for split and high bandwidth isochronous transactions (see 5.92, 11.20, and 11.21 for more information)
Handshake	ACK	0010B	Receiver accepts error free data packet
	NAK	1010B	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110B	Endpoint is halted or a control pipe request is not supported
	NYET	0110B	No response yet from receiver (see 8.5.1 and 11.7–11.21)
Special	PRE	1100B	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	ERR	1100B	(Handshake) Split transaction error handshake (reuses PRE value)
	SPLIT	1000B	(Token) High-speed split transaction token (see 8.4.2)
	PING	0100B	(Token) High-speed flow control probe for a bulk/control endpoint (see 8.5.1)
	Reserved	0000B	Reserved PID

2.3 Frames, Transfers, and Transactions

Packets are grouped together into transactions. Each transaction has up to three phases, or parts that occur in sequence: token, data, and handshake. In the token phase, the host sends out a token packet. Depending on the type of transaction, the data phases and handshake phases are optional. Transfers are groups of transactions. Each transfer contains one or more transactions. Transfers are grouped into frames. Each frame begins with a (start-of-frame) SOF packet. Full-speed frames are 1 ms wide.

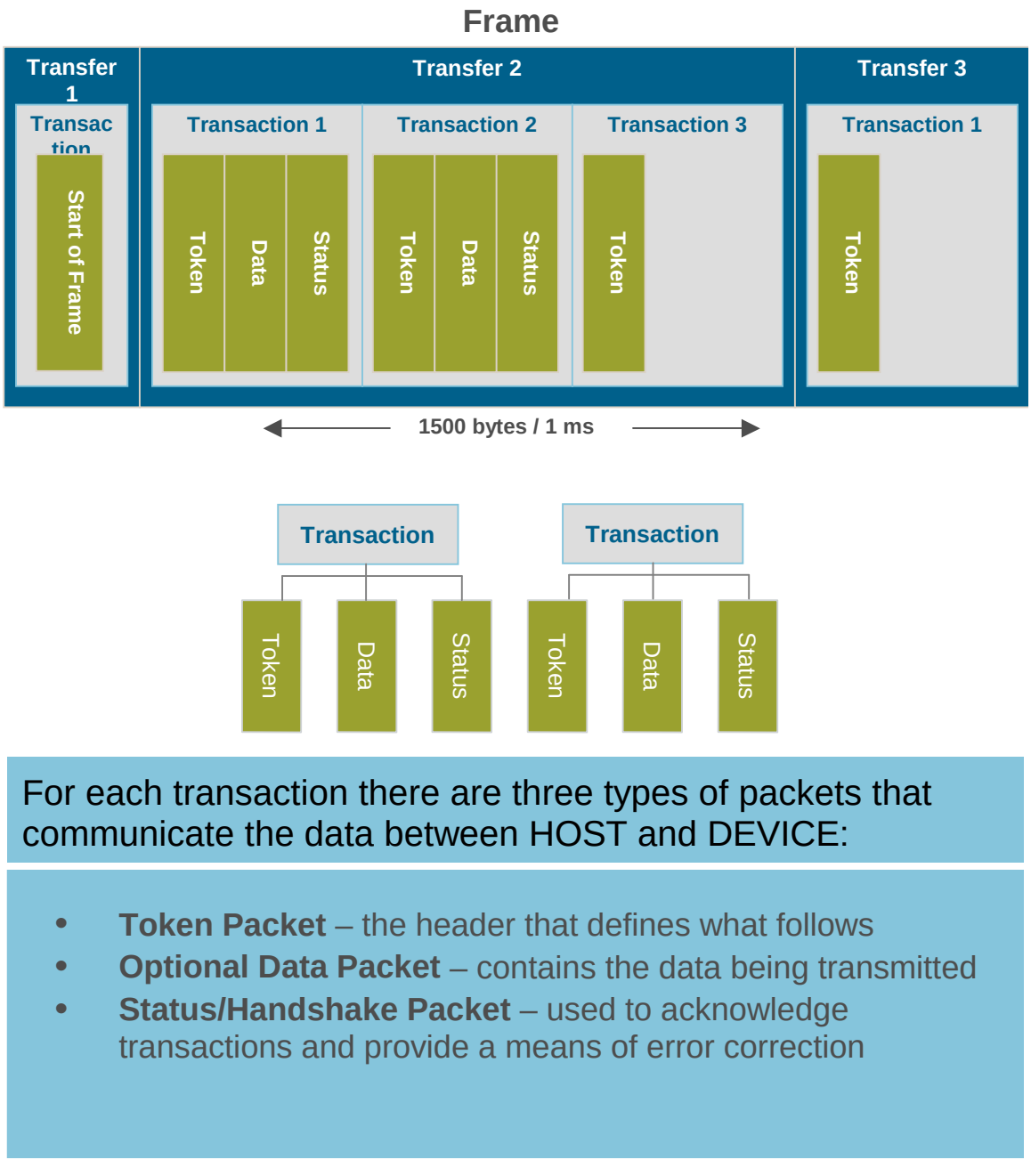


Figure 3. A USB Frame

2.4 The Data Toggle Bit

When a transfer contains multiple transactions, a data-toggle bit is used to keep the transmitting and receiving devices synchronized. The data-toggle bit is included in the PID field in the token packet. When the device is configured, the data-toggle bit is set to 0. When the receiver detects an incoming data transaction, it compares the received data-toggle bit to the state of its own data-toggle. If the bits match, the receiver toggles its bit and returns a ACK handshake packet to the sender. Upon receiving the ACK, the sender toggles its bit. The data-toggle bit is a 1 bit sequence number.

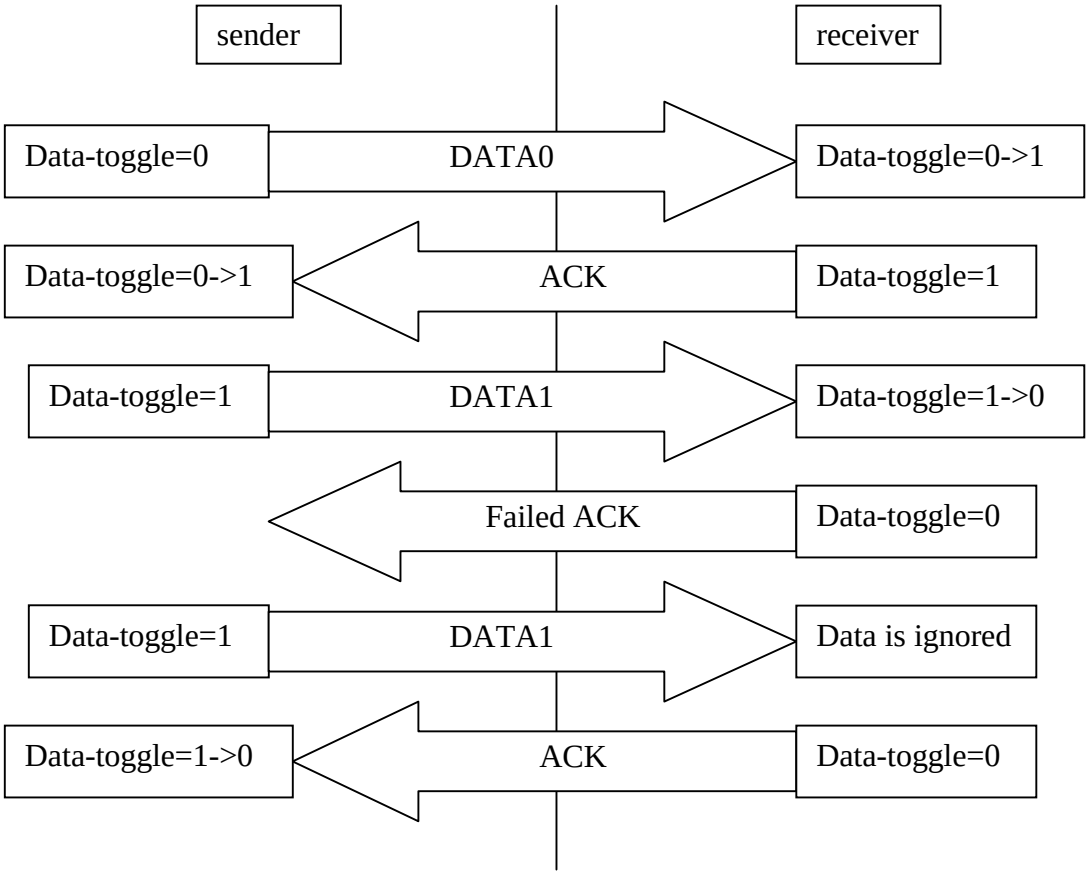


Figure 4. Toggle Bit Sequence

2.5 Transaction Types

There are three transaction types: IN, OUT, and Setup. The transaction type is determined by the PID in the token packet. Because only the host sends token packets, the host determines the transaction type.

Table 2. Transaction Types

Transaction Type	Source of Data	Type of Transfers That Uses This Transaction Type	Contents
IN	Device	All	Generic data
OUT	Host	All	Generic data
Setup	Host	Control	A request

2.6 Transfer Types

Transactions are grouped into transfers. There are four types of transfers: control, bulk, interrupt, and isochronous. Each transfer is optimized for a specific type of data transfer.

Table 3. Transfer Types

Transfer Type	Stages (Transactions)	Phases (Packets)
Control	Setup	Token
		Data
		Handshake
	Data (IN or OUT) (optional)	Token
		Data
		Handshake
	Status (IN or OUT)	Token
		Data
		Handshake
Bulk	Data (IN or OUT)	Token
		Data
		Handshake
Interrupt	Data (IN or OUT)	Token
		Data
		Handshake
Isochronous	Data (IN or OUT)	Token
		Data

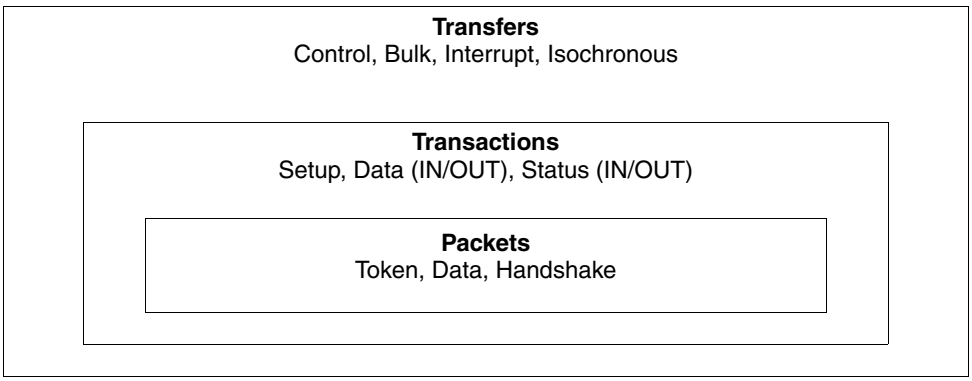


Figure 5. Putting It All Together

2.7 Control Transfer

The host sends commands and query parameters via control packets. All devices use control transfers to Endpoint 0 for the enumeration process.

The maximum transfer rate is 832 Kbytes/second for full-speed operation.

The maximum size for a control packet is 8, 16, 32, or 64 bytes for full-speed operation. These sizes are data only. All packets except the last one, must be the maximum packet size. The maximum packet size is determined during the enumeration phase. It is located in the device descriptor for the default control pipe (Endpoint 0), and the endpoint descriptor for all other pipes. If a transfer has more data then will fit in one transaction, the host sends or requests the data in multiple transaction.

It should be noted on the following diagrams, that stall and NAK handshake packets can be sent during the data phase.

Control transfers can have up to three stages, each stage starting with its own token packet. When not in one of these three stages, the transfer is in the idle state.

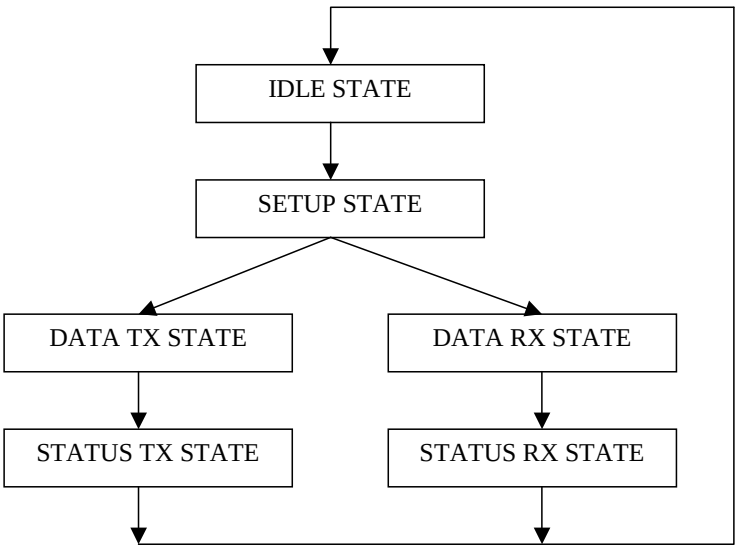


Figure 6. Control Transfer State Machine

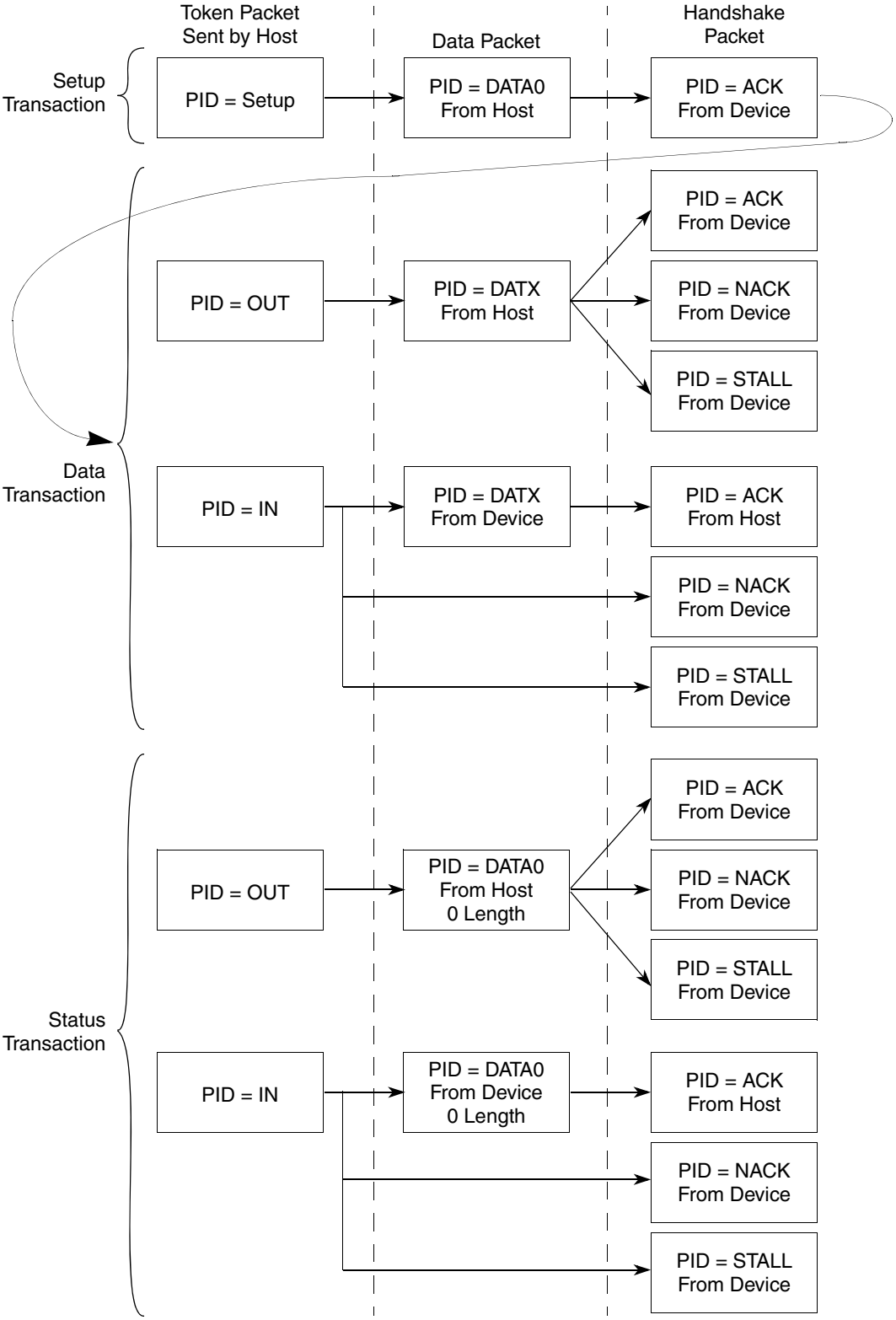


Figure 7. Control Transfer

2.8 Interrupt Transfers

Interrupt transfers occur periodically, with a period specified by the endpoint descriptor. The interrupt transfer is different from the isochronous transfer in that the period is outside the time of a single frame. There is not an interrupt mechanism in USB for devices. A device must always wait for a poll from the host to send data. The maximum full-speed data rate is 64 KB per second. The time specified in the endpoint descriptor is a maximum time. The host may poll for data at a faster rate. Because of this, there is no guaranteed transfer rate; it may be faster then required, but not slower.

If the host polls the device, and the device has no data to send, it responds with a NAK. Interrupt transfers contain only one transaction. Interrupt transfers are handshaked.

Bulk transfers are the same as interrupt transfers. The difference depends on how the host allocates bus bandwidth. Bulk transfers defer bandwidth to all other transfer types.

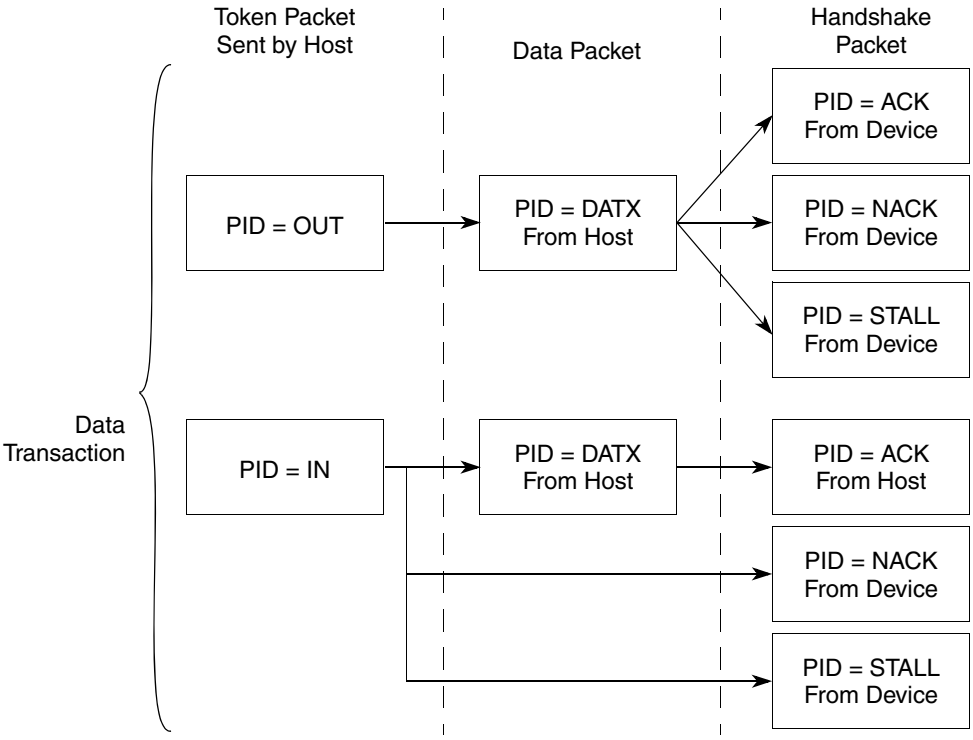


Figure 8. Interrupt Transfers

2.9 Isochronous Transfers

Isochronous transfers guarantee a fixed transfer rate. Data is polled/sent on every frame at a fixed size. A full-speed isochronous transaction can send 1023 bytes per frame. A frame is sent every 1 ms. This translates to 1.023 megabytes per second.

The minimum is 1 byte per frame or 1 kilobyte per second. Isochronous transactions are not handshaked. Isochronous transfers contain only one transaction.

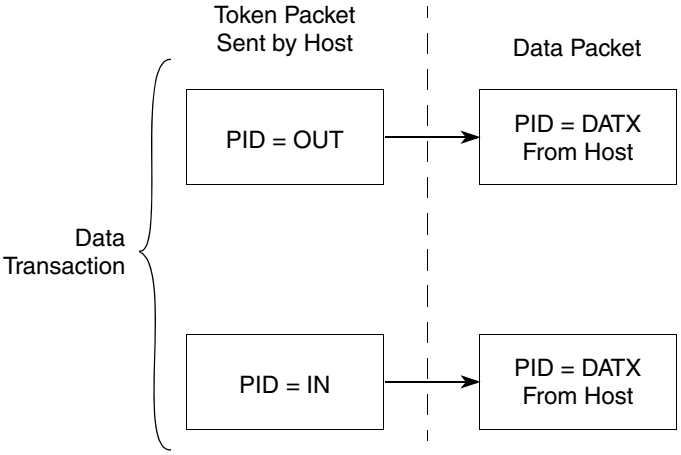


Figure 9. Isochronous Transfers

2.10 The Start-Of-Frame (SOF) Packet

For full-speed operation, the host sends a SOF packet every 1 ms, indicating the start of a frame. The SOF packet provides a time reference for use by synchronous devices and hubs. Because the host initiates all transactions by sending a token packet, the device actually synchronizes to the token packet for normal data transfers, the SOF packet can be ignored.

The SOF packet is also used as a keep-alive mechanism. As long as the host is sending SOF packets to the device, the device cannot go into a suspend mode.

3 The USB OTG Capable Controller Module

This section describes the USB Dual-Mode (DM) controller. The On-The-Go (OTG) implementation in this module provides limited host functionality and device FS solutions for implementing a USB 2.0 full-speed/low-speed compliant peripheral. The OTG implementation supports the On-The-Go (OTG) addendum to the USB 2.0 Specification. Only one protocol can be active at any time. A negotiation protocol must be used to switch to a USB host functionality from a USB device. This is known as the master negotiation protocol (MNP).

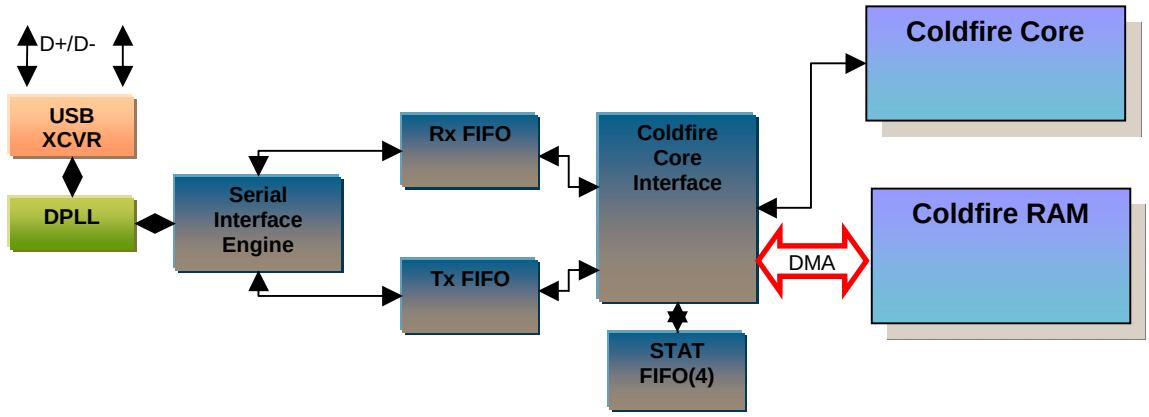


Figure 10. USB OTG Module Block Diagram

The module supports 16 bidirectional endpoints, DMA or FIFO data stream interfaces, and OTG protocol logic. The USB stack communicates with the hardware via buffer descriptors and EHCI compliant register set.

3.1 EHCI Registers

The enhanced host controller interface (EHCI) specification describes the register-level interface for a host controller for the *Universal Serial Bus (USB) Revision 2.0*. The EHCI specification defines two sets of registers:

Capability registers	Specify the limits, restrictions, and capabilities of a host/device controller implementation.
Operational registers	Used by the system software to control and monitor the operational state of the host controller.

3.2 Interrupt Status Register

The Interrupt Status register is a critical part of the interface to the OTG module.

	7	6	5	4	3	2	1	0
R W	STALL	ATTACH	RESUME	SLEEP	TOK_DNE	SOF_TOK	ERROR	USB_RST
Reset:	0	0	0	0	0	0	0	0

Figure 11. Interrupt Status Register

Table 4. Interrupt Status Register Field Descriptions

Field	Description
7 STALL	Stall Interrupt In Target mode this bit is asserted when a STALL handshake is sent by the SIE. In Host mode this bit is set when the USB Module detects a STALL acknowledge during the handshake phase of a USB transaction. This interrupt can be use to determine is the last USB transaction was completed successfully or if it stalled.
6 ATTACH	Attach Interrupt This bit is set when the USB Module detects an attach of a USB device. This signal is only valid if HOST_MODE_EN is true. This interrupt signifies that a peripheral is now present and must be configured.
5 RESUME	This bit is set depending upon the DP/DM signals, and can be used to signal remote wake-up signaling on the USB bus. When not in suspend mode this interrupt should be disabled.
4 SLEEP	This bit is set when the USB Module detects a constant idle on the USB bus for 3 milliseconds. The sleep timer is reset by activity on the USB bus.
3 TOK_DNE	This bit is set when the current token being processed has completed. The ColdFire core should immediately read the STAT register to determine the EndPoint and BD used for this token. Clearing this bit (by writing a one) causes the STAT register to be cleared or the STAT holding register to be loaded into the STAT register.
2 SOF_TOK	This bit is set when the USB Module receives a Start Of Frame (SOF) token. In Host mode this bit is set when the SOF threshold is reached, so that software can prepare for the next SOF.
1 ERROR	This bit is set when any of the error conditions within the ERR_STAT register occur. The ColdFire core must then read the ERR_STAT register to determine the source of the error.
0 USB_RST	This bit is set when the USB Module has decoded a valid USB reset. This informs the Microprocessor that it should write 0x00 into the address register and enable endpoint 0. USB_RST is set after a USB reset has been detected for 2.5 microseconds. It is not asserted again until the USB reset condition has been removed and then reasserted.

3.3 DATA Transfers

All USB OTG module interface/control registers are 8 bits wide. This allows the module to scale across the 8-bit and 32-bit parts. Registers that require 32 bits (pointers) are split up into four separate registers. An example of this is the buffer descriptor table (BDT) page register. The BDT page register is initialized with a pointer to the start of the buffer descriptor table. The buffer descriptor table is a table of buffer descriptors in memory, aligned to a 512 byte boundary.

3.4 How the Pointer into the BDT is Calculated

31:24	23:16	15:9	8:5	4	3	2:0
BDT_PAGE3	BDT_PAGE2	BDT_PAGE1	ENDPOINT	IN	ODD	000

Figure 12. BDT Pointer

BDT_PAGE	BDT_PAGE registers in the control register block
END_POINT	END_POINT field from the USB TOKEN TX 1 for an TX transmit transfers and 0 for an RX receive transfers
ODD	This bit is maintained within the USB-FS SIE. It corresponds to the buffer that is currently in use. The buffers are used in a ping-pong fashion.
000	Each buffer descriptor is 8 bytes.

Data transfers between endpoints and RAM are configured via buffer descriptors. A buffer descriptor contains a 32 bit status/control entry and a 32 bit pointer to a buffer in RAM (8 bytes). For each endpoint, there are four buffer descriptors. Every endpoint direction requires two 8-byte buffer descriptor entries. Therefore, a system with 16 fully bidirectional endpoints requires 512 bytes of system memory to implement the BDT. The two buffer descriptor (BD) entries allows for an EVEN BD and ODD BD entry for each endpoint direction. This allows the microprocessor to process one BD while the USB-FS is processing the other BD. Double buffering BDs in this way allows the USB-FS to easily transfer data at the maximum throughput provided by USB.

31:26	25:16	15:8	7	6	5	4	3	2	1	0
RSVD	BC (10-Bits)	RSVD	OWN	DATA0/1	KEEP/ TOK_PID[3]	NINC/ TOK_PID[2]	DTS/ 5 TOK_PID[1]	BDT_STALL/ TOK_PID[0]	0	0
Buffer Address (32-Bits)										

Figure 13. Buffer Descriptor

The buffer address entry points to the start of the buffer in RAM. The BC bits (10 bits) set the length of the buffer. The OWN bit is a semaphore mechanism used to indicate whether the CPU core or the USB module own the buffer descriptor. The USB module will not modify or use the buffer descriptor if OWN is zero. The software should only modify the buffer descriptor when OWN is zero. After the software sets up the buffer descriptor, the last thing it should do is set OWN to 1.

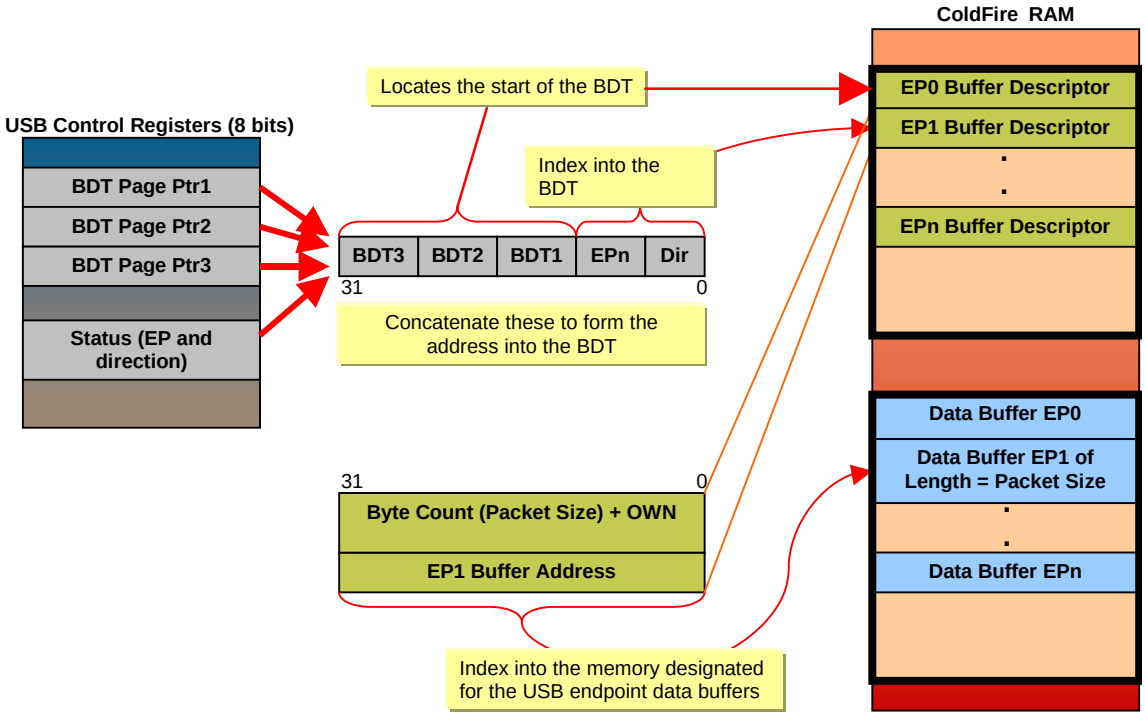


Figure 14. Buffer Descriptor Table (512 Bytes Max)

3.5 Firmware Example

Setting up the buffer descriptor table:

The following code should be inserted in the link command file to create a 512 byte aligned area in RAM.

```
/* Buffer descriptor base address
   shall be aligned to 512 byte boundary.
   Size shall be 512 bytes. */
                  = ALIGN(512);
 __BDT_BASE       = .;
                  = . + 512;
 __BDT_END        = .;
```

Then the BDT_PAGE registers can be initialized from C code using the following macros and code snippet.

```
/* This macro shall evaluate to a uint32 pointer to the start address of the
   buffer descriptor table (BDT). The BDT has 32 bytes for each endpoint.
   The BDT shall be aligned to 512 byte boundary! */

extern hcc_u32 _BDT_BASE[];

#define BDT_BASE             ((hcc_u32*)(_BDT_BASE))

/* Set BDT address. */
MCF_USB_BDT_PAGE_01 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 8);
MCF_USB_BDT_PAGE_02 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 16);
MCF_USB_BDT_PAGE_03 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 24);
```

4 Using the USB OTG Module in Host Mode

The USB OTG module makes developing host-side USB drivers easy. The hardware does all the work. The hardware does so much of the work that a host driver can be implemented without interrupts. The hardware supports features such as automatic SOF packet generation and TX, device detection, and auto transaction transfers.

Host mode allows bulk, isochronous, interrupt, and control transfers. Bulk data transfers are performed at nearly the full USB bus bandwidth. Support is provided for ISO transfers, but the number of ISO streams that can be practically supported is affected by the interrupt latency of the processor servicing the token during interrupts from the SIE.

4.1 Device Detection and Speed Determination

USB communications occur over 2 wires (D+ and D–). The host pulls these wires low using 2 pulldown resistors. A device has a pullup on either the D+ or D– line depending on the speed it supports (full or low). The host can detect a device being plugged in by sensing when either the D+ or D– lines go high. It can also determine the speed using this system.

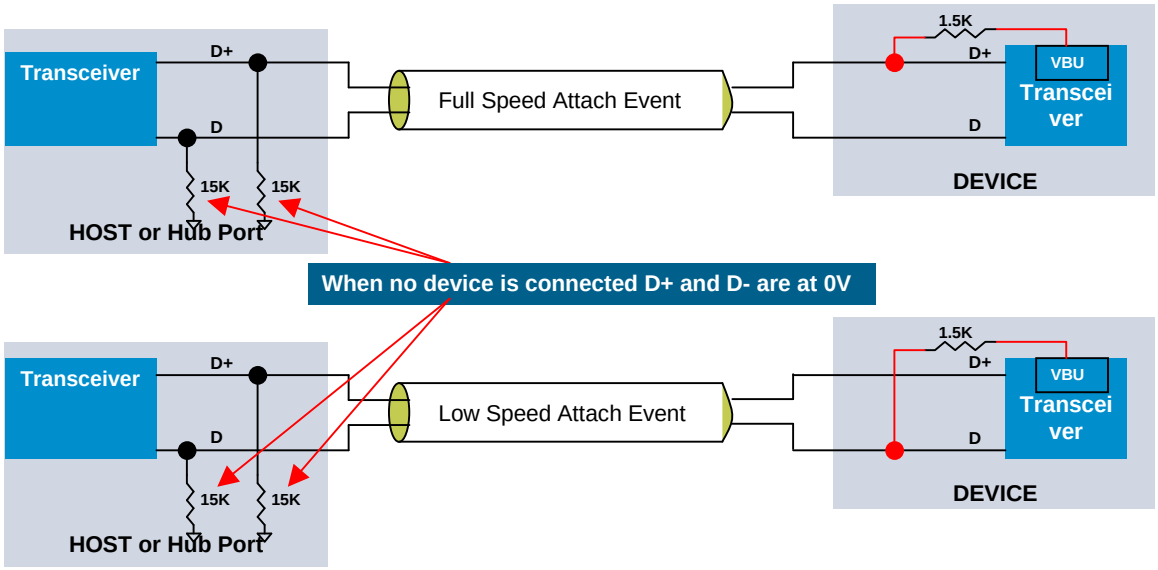


Figure 15. The Physical Layer

When a device is plugged in, the USB OTG controller generates a ATTACH indication or interrupt if the interrupt is enabled.

Table 5. Attach Interrupt Register (ATTACH)

6 ATTACH	Attach Interrupt This bit is set when the USB Module detects an attach of a USB device. This signal is only valid if HOST_MODE_EN is true. This interrupt signifies that a peripheral is now present and must be configured.

The firmware can determine the speed of the device via the state of the JSTATE and SE0 bits in the control register. If the JSTATE bit is 0, then the connecting device is low speed.

4.2 The Start-Of-Frame Packet

The host sends a SOF packet every 1 ms (full speed) to keep the device from entering suspend mode. The USB OTG module generates this packet automatically. The SOF threshold register is used to program the number of USB byte times before USB controller stops initiating token packet transaction. This register must be set to a value that ensures that other packets are not actively being transmitted when the SOF time counts to zero. When the SOF counter reaches the threshold value, no more tokens are transmitted until after the SOF has been transmitted. The value programmed into the threshold register must reserve enough time to ensure the worst case transaction completes. In general, the worst case transaction is a IN token followed by a data packet from the target followed by the response from the host. The actual time required is a function of the maximum packet size on the bus. Typical values for the SOF threshold are: 64-byte packets=74; 32-byte packets=42; 16-byte packets=26; 8-byte packets=18.

Field	PID	Frame Number	CRC
Bits	8	11	5

Figure 16. SOF Packet

The SOF packet contains a 11 bit frame number. The USB OTG module automatically increments the frame counter. The counter can be reset or read by the firmware via the FRM_NUML and FRM_NUMH registers.

4.3 Addressing a Device

USB support up to 127 devices on a node. This is accomplished by an address field in the token packet.

Field	PID	Address	Endpoint	CRC
Bits	8	7	4	5

Figure 17. Token Packet

When operating in Host mode (HOST_MODE_EN=1), the USB module transmits this address with a TOKEN packet. This enables the USB module to uniquely address an USB peripheral. In either mode, the USB_EN bit within the control register must be set. The Address register is reset to 0x00 after the reset input becomes active or the USB module decodes a USB reset signal. This action initializes the Address register to decode address 0x00 as required by the USB specification.

Table 6. Address Register (ADDR)

Field	Description
7 LS_EN	Low Speed Enable bit. This bit informs the USB Module that the next token command written to the token register must be performed at low speed. This enables the USB Module to perform the necessary preamble required for low-speed data transmissions.
6–0 ADDR	USB address. This 7-bit value defines the USB address that the USB Module decodes in peripheral mode, or transmit when in host mode.

4.4 Data Transactions in Host Mode

The host starts all transactions by sending a token packet. Like device mode operation, the USB OTG module uses buffer descriptors in a buffers descriptor table to transmit and receive data. The hardware handles most of the transaction. A token packet is created by the USB OTG module using data from the TOKEN register. For an OUT transaction, the host then automatically sends the DATA packet. Finally, the hardware verifies the ACK from the device and automatically resends the data if the transaction is not ACKed.

With the hardware doing so much of the work, the USB host-side driver can be a very simple piece of software.

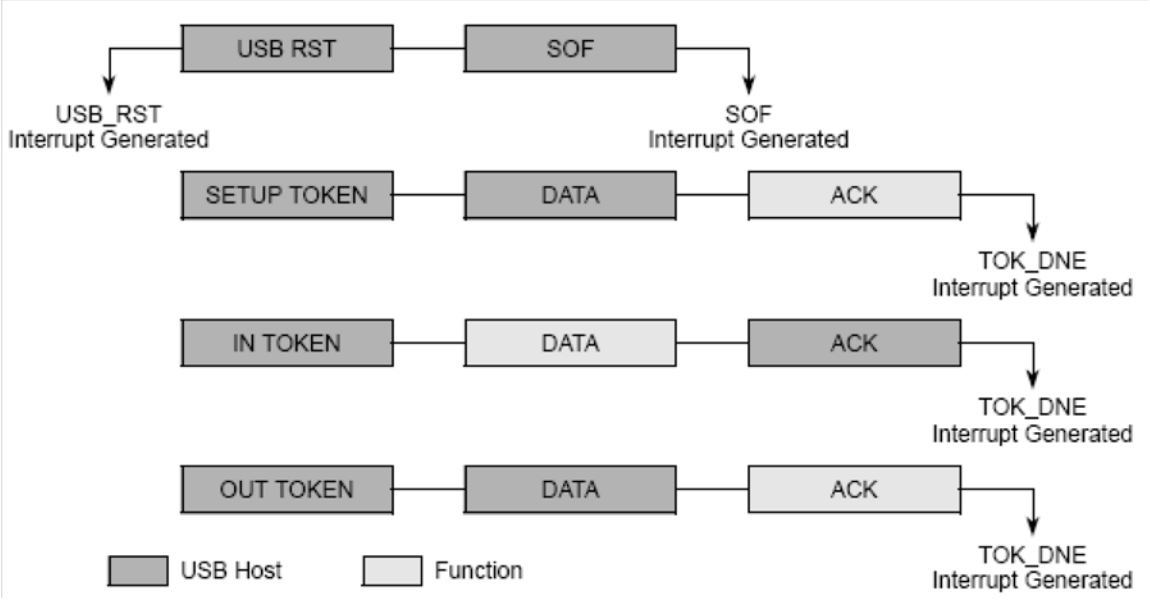


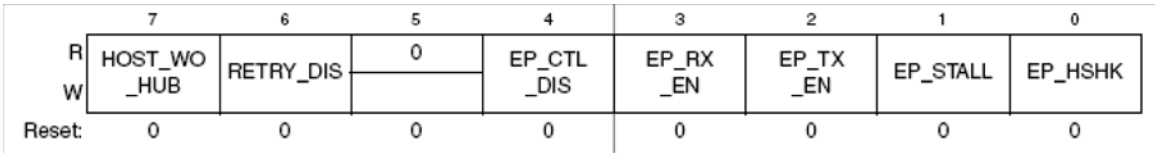
Figure 18. USB Host-Side Driver

4.5 Endpoint Control Registers

The endpoint control registers contain the endpoint control bits for each of the 16 endpoints available within the USB module for a decoded address. The format for these registers is shown in the following figure. Endpoint 0 (ENDPT0) is associated with control pipe 0 which is required for all USB functions. Therefore, after a USB_RST interrupt occurs the ColdFire core should set the ENDPT0 register to contain 0x0D.

In Host mode, ENDPT0-15 is used to determine the handshake, retry, and low-speed characteristics of the host transfer. For control, bulk, and interrupt transfers in Host mode, the EP_HSHK bit must be set to 1. For isochronous transfers, it must be set to 0. Common values to use for ENDPT0-15 in host mode are 0x4D for control, bulk, and interrupt transfers, and 0x4C for isochronous transfers.

There are 16 endpoint registers ENDPT0-15.



	7	6	5	4	3	2	1	0
R W	HOST_WO_HUB	RETRY_DIS	0	EP_CTL_DIS	EP_RX_EN	EP_TX_EN	EP_STALL	EP_HSHK
Reset:	0	0	0	0	0	0	0	0

Figure 19. Endpoint Register (0–15)

Table 7. Endpoint Register (ENDPT0–15)

Field	Description
7 HOST_WO_HUB	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit allows the host to communicate to a directly connected low speed device. When cleared, the host produces the PRE_PID then switch to low speed signaling when sending a token to a low speed device as required to communicate with a low speed device through a hub.
6 RETRY_DIS	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit causes the host to not retry NAK'ed (Negative Acknowledgement) transactions. When a transaction is NAKed, the BDT PID field is updated with the NAK PID, and the TOKEN_DNE interrupt is set. When this bit is cleared NAKed transactions is retried in hardware. This bit must be set when the host is attempting to poll an interrupt endpoint.
4 EP_CTL_DIS	This bit, when set, disables control (SETUP) transfers. When cleared, control transfers are enabled. This applies if and only if the EP_RX_EN and EP_TX_EN bits are also set.
3 EP_RX_EN	This bit, when set, enables the endpoint for RX transfers
2 EP_TX_EN	This bit, when set, enables the endpoint for TX transfers
1 EP_STALL	When set this bit indicates that the endpoint is called. This bit has priority over all other control bits in the EndPoint Enable Register, but it is only valid if EP_TX_EN=1 or EP_RX_EN=1. Any access to this endpoint causes the USB Module to return a STALL handshake. After an endpoint is stalled it requires intervention from the Host Controller.
0 EP_HSHK	When set this bet enables an endpoint to perform handshaking during a transaction to this endpoint. This bit is generally set unless the endpoint is Isochronous.

5 Introduction to the CMX Stack

The CMX USB-Lite stack can be downloaded from the Freescale web site. The firmware is provided as public source under the license included in the installation executable.

5.1 Included Projects

HID keyboard device	The USB stack emulates a keyboard.
HID mouse device	The USB stack emulates a mouse.
HID generic device	Implements a user defined HID device
HID PC software (VC++)	Used to communicate with the HID generic device.
HID host	Allows the user to connect and communicate with a keyboard, mouse, or joystick
CDC device, serial to USB	The USB stack can be used as a USB to serial converter.
Mass-Storage host	Using this firmware, the USB stack can read and write to a USB flash stick.

5.2 The Files in the Firmware

USB Basic device controller

Usb.c	Low-level USB driver and usb_it_handler() isr
Usb.h	Header for USB driver
Usb_config.h	USB driver compile time configuration parameters

HID device class files

Hid.c	HID device layer including the state machine
Hid.h	Header for hid.c
Hid_generic.c	Demo application using a generic report structure Used to communicate with the board via the HID PC application
Hid_generic.h	Header for hid_generic.c
Hid_kbd.c	HID keyboard demo
Hid_kbd.h	
Hid_mouse.c	HID mouse demo
Hid_mouse.h	
Hid_usb_config.c	USB enumeration structures for keyboard, mouse, and generic HID demos
Hid_usb_config.h	Header for hid_usb_config.c
Hid_main.c	Initial entry point for HID device demos

CDC class files

Cdc_usb_config.c	USB configuration structures for the serial to USB CDC demo
Cdc_usb_config.h	Header file for cdc_usb_config.c

Cdc_main.c	Main loop, read/writes the UART, and transfers data to the USB stack
Usb_cdc.c	CDC layer for the USB driver, cdc_getch() and cdc_putch()
Usb_cdc.h	Header for usb_cdc.c
USB basic host controller	
Usb_host.c	USB host driver
Usb_host.h	Header for usb_host.c
Usb_utils.c	USB host driver utility functions
Usb_utils.h	Header file for usb_utils.c
HID host class files	
Hid_demo.c	Prints information about connected devices to the serial line
Hid_parser.c	Read and write access to HID report items
Hid_parser.h	Header file for hid_parser.c
Hid_usage.h	HID usage codes
Host_hid.c	Basic HID functionality used by HID drivers
Host_hid.h	Header for host_hid.c
Host_hid_joy.c	HID joystick driver
Host_hid_kbd.c	HID keyboard driver
Host_hid_mouse.c	HID mouse driver
Mass storage host	
Mst_main.c	Initial entry point for mass storage demo
Scsi.c	SCSI layer for USB mass storage class host demo
Usb_mst.c	USB layer of USB mass storage class host driver
Terminal.c	Simple terminal interface allows users to execute commands to exercise the mass storage stack
Mst_glue.h	Links sector driver of FAT file system to SCSI Layer of mass storage driver
Thin-lib.a	Limited function FAT file system library

6 Enumeration

Enumeration is the process of the host learning about the device. Enumeration occurs on Endpoint 0, using default address 0. During the enumeration process, the host assigns another address to the device. All USB devices must support control transfers, the standard requests, and Endpoint 0. For a successful enumeration, the device must respond to each request by returning requested information and taking other requested actions.

The USB specification defines six device states. During enumeration, a device moves through four of the states: powered, default, address, and configured. (The other states are attached and suspend.)

Attached: The device can be in an attached or detached stat; the specification only defines the attached stat.

Powered: A USB device must be able to be addressed within a specified time period from when power is initially applied. After an attachment to a port has been detected, the host may enable the port, which will also reset the device attached to the port. The host will provide up to 100 ma at this point.

Default: After the device has been powered, it must not respond to any bus transactions until it has received a reset from the bus. After receiving a reset, the device is then addressable at the default address (0).

Address: All USB devices use the default address when initially powered or after the device has been reset. Each USB device is assigned a unique address by the host after attachment or after reset. A USB device maintains its assigned address while suspended. A USB device responds to requests on its default pipe whether the device is currently assigned a unique address or is using the default address.

Configured: Before a USB device's function may be used, the device must be configured. Configuration involves correctly processing a SetConfiguration() request with a non-zero configuration value. Configuring a device or changing an alternate setting causes all of the status and configuration values associated with endpoints in the affected interfaces to be set to their default values. This includes setting the data toggle of any endpoint using data toggles to the value DATA0.

Suspended: To conserve power, USB devices automatically enter the suspended state when the device has observed no bus traffic for a specified period (refer to Section 7, "The USB Device-Side Driver,"). When suspended, the USB device maintains any internal status, including its address and configuration. All devices must suspend if bus activity has not been observed for the length of time specified in the USB specification. Attached devices must be prepared to suspend at any time they are powered, whether they have been assigned a non-default address or are configured.

The enumeration process is driven by the host. The host submits USB level requests to the device, the device responds with a action or data.

6.1 Enumeration Steps

1. The device is plugged into a host. The host provides power to the device with a current limit of 100 ma.
2. The host determines low-speed/full-speed capability by pullup resistors connected to either the D+ or D– lines. At this point, the device is in the powered state.
3. The host sends a reset to the device by setting D+ and D– low for at least 10 milliseconds. When the host removes the reset, the device goes into the default state.

4. In the default state, the device is ready to respond to control transfers at Endpoint 0. The host communicates with the device using the default address of 00. The device can draw up to 100 ma from the host.
5. The host sends a GET_DESCRIPTOR request to Endpoint 0, address 0, to learn the maximum packet size of the default pipe. The eight-byte device descriptor contains the maximum packet size supported by Endpoint 0.
6. The host assigns a unique address to the device by sending a SET_ADDRESS request. The device is in the address state.
7. The host sends a GET_DESCRIPTOR request to the new address to read the full device descriptor.
8. The host then requests any additional descriptors specified in the device descriptor. Each descriptor begins its length and type.
9. The host assigns a device driver based on the data in the descriptors. Windows® will use the devices vendor ID and product ID to find an appropriate INF file to determine what drivers to load. If there is no match, Windows will use a default driver according to class.
10. If the device supports multiple configurations, the host will send a SET_CONFIGURATION request to the device to select the desired configuration.

6.2 Types of Descriptors

Type	Value	Descriptor
Standard	01	device
	02	configuration
	03	string
	04	interface
	05	endpoint
	06	device qualifier
	07	other speed configuration
	08	interface power
Class	21	HID
	29	hub
Specific to HID	22	report
	23	physical

6.3 Device Descriptor

Because the device descriptor represents the entire device, there can be only one per device. This descriptor specifies basic information such as: USB version supported, maximum packet size, the number of configurations, and vendor and product ID info.

Table 8. Device Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	Device Descriptor Type(0x01)
2	bcdUSB	2	BCD	USB Specification Release Number 0x0200

Table 8. Device Descriptors (continued)

Offset	Field	Size	Value	Description
4	bDeviceClass	1	Class	Class code (see spec)
5	bDeviceSubClass	1	SubClass	Subclass code (see spec)
6	bDeviceProtocol	1	Protocol	Protocol code (see spec)
7	bMaxPacketSize0	1	Number	Max packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID
10	idProduct	2	ID	Product ID
12	bcdDevice	2	ID	Device release number in BCD
14	iManufacturer	1	BCD	Index of string descriptor describing manufacturer
15	iProduct	1	Index	Index of string descriptor describing product
16	iSerialNumber	1	Index	Index of string descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

6.4 Configuration Descriptor

When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned. Each device has at least one configuration descriptor that describes the device's features and abilities. The device can have multiple configurations. The host selects a specific configuration using a SET_CONFIGURATION request.

Table 9. Configuration Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	Configuration descriptor type (0x02)
2	wTotalLength	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class or vendor specific), returned for this configuration.
4	bNumInterfaces	1	Number	Number of interfaces supported by this configuration
5	bConfigurationValue	1	Number	Value to use as an argument to themSetConfiguration() request to select this configuration
6	iConfiguration	1	Index	Index of string descriptor describing this configuration
7	bmAttributes	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-poweredD5: Remote Wakeup D4...0: Reserved (reset to zero)
8	bMaxPower	1	mA	Maximum power consumption of the USB device from the bus in this Specific configuration when the device is fully operational. Expressed in 2 mA units (i.e., 50 = 100 mA).

6.5 Interface Descriptor

A configuration's interface descriptor contains information about the endpoints the interface supports. Each configuration must support at least one interface, but can support many. The host selects the interface with a SET_INTERFACE request.

Table 10. Interface Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	Interface descriptor type (0x04)
2	bInterfaceNumber	1	Number	Number of this interface.
3	bAlternateSetting	1	Number	Value used to select this alternate setting for the interface identified in the prior field
4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses the default control pipe.
5	bInterfaceClass	1	Class	Class code
6	bInterfaceSubClass	1	SubClass	Subclass code
7	bInterfaceProtocol	1	Protocol	Protocol code If this field is reset to zero, the device does not use a class-specific protocol on this interface. If this field is set to FFH, the device uses a vendor-specific protocol for this interface.

6.6 Endpoint Descriptor

Every endpoint except 0 must have a endpoint descriptor.

Table 11. Endpoint Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	Endpoint descriptor type(0x05)
2	bEndpointAddress	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint

Table 11. Endpoint Descriptors (continued)

Offset	Field	Size	Value	Description
4	wMaxPacketSize	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For all endpoints, bits 10..0 specify the maximum packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None(1 transaction / microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero.
6	bInterval	1	Number	Interval for polling endpoint for data transfers. Expressed in frames or microframes depending on the device operating speed (i.e., either 1 millisecond or 125 s units). For full-/high-speed isochronous endpoints, this value must be in the range from 1 to 16. The bInterval value is used as the exponent for a 2bInterval-1 value; e.g., a bInterval of 4 means a period of 8 (24-1). For full-/low-speed interrupt endpoints, the value of this field may be from 1 to 255. For high-speed interrupt endpoints, the bInterval value is used as the exponent for a 2bInterval-1 value; e.g., a bInterval of 4 means a period of 8 (24-1). This value must be from 1 to 16. For high-speed bulk/control OUTendpoints, the bInterval must specify the maximum NAK rate of the endpoint. A value of 0 indicates the endpoint never NAKs. Other values indicate at most 1 NAK each bInterval number of microframes. This value must be in the range from 0 to 255.

6.7 String Descriptor

The string descriptor contains descriptive text. The specification supports string descriptors for: the manufacture, product, serial number, configuration, and interface. String descriptors are optional.

Table 12. String Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	String descriptor type (0x03)
2	bString	N	Number	Unicode encoded string

The bString field is overloaded. For string descriptor 0, the bString is an array of 1 or more language identifier codes. For other string descriptors, this is a unicoded string.

Unicode uses 2 bytes per character.

6.8 Common Descriptor Hierarchy

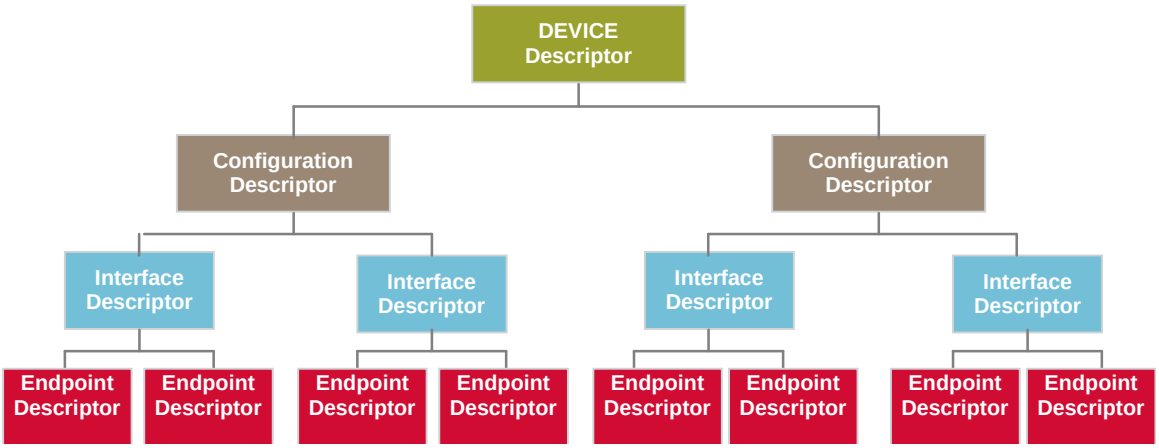


Figure 20. Common Descriptor Hierarchy

6.9 USB Standard Device Requests

The USB specification defines 11 standard requests for control transfer. These requests are used by the host to enumerate the device. During enumeration, the hosts sends these requests to the device after the device is reset, starting when it is in the default state. The device firmware simply responds the to requests with either data or an action.

Table 13. USB Standard Device Requests

Request#	Name	Data Source	Value	Data Length	Data
0x00	Get_Status	device	0	2	status
0x01	Clear_Feature	none	feature	0	none
0x03	Set_Feature	none	feature	0	none
0x05	Set_Address	none	address	0	none
0x06	Get_Descriptor	device	type&index	x	descriptor
0x07	Set_Descriptor	host	type&index	x	descriptor
0x08	Get_Configuration	device	0	1	configuration
0x09	Set_Configuration	none	config	0	none
0x0A	Get_Interface	device	0	1	alternate
0x0B	Set_Interface	none	interface	0	none
0x0C	Sync_Frame	device	0	2	frame #

The data source parameter defines who is sending the data. The value column specifies the data in the value field. The data column specifies the data transferred during the data stage of the USB transaction. The data length specifies the number of bytes in the data field. The *x* specifies the descriptor length.

6.10 USB Standard Device Requests—Setup Transfer Data Packet Data Format

The 11 standard device requests are sent from the host on the devices default control pipe. These requests are made using control transfers. Control transfers have three phases: setup, data, and handshake. The setup transaction includes a data packet sent from the host to the device (out transfer). The request is transferred in the data packet of the setup transaction.

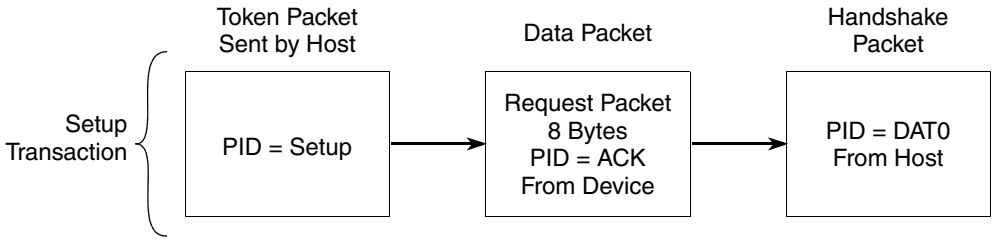


Figure 21. Setup Transaction on Control Transfer

Table 14. Device Request

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6 . . . 5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4 . . . 0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4 . . . 31 = Reserved
1	bRequest	1	Request #	Specific request
2	wValue	2	Value	Word-sized field that varies according to request
4	wIndex	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

6.11 Decoding the Generic HID Descriptors

```
#define USB_FILL_DEV_DESC(usb_ver, dclass, dsubclass, dproto, psize, vid,\
                          pid, relno, mstr, pstr, sstr, ncfg)

hcc_u8 geh_device_descriptor[] = {
USB_FILL_DEV_DESC(
     0x0101,        // USB version 1.1
     0,             // device class 0 ( each interface defines it's own class info )
     0,             // device subclass 0 ( each interface defines it's own class info )
     0,             // device protocol  0 ( does not use class-specific protocols )
EP0_PACKET_SIZE,    // endpoint 0 control packet size is 8 bytes
GEH_VENDOR_ID,      // Vendor ID = 0xC1CA
GEH_PRODUCT_ID,     // Product ID is 3
GEH_DEVICE_REL_NUM, // Release number 0
       1,            // String index 1 is manufacture name
       2,            // String index 2 is Product name
       3,            // String index 3 is Serial Number
       1)            // Only 1 configuration

hcc_u8 * geh_string_descriptors[] = {
                                     string_descriptor0, // String index 0 placeholder
                                     str_manufacturer,   // String index 1
                                     geh_product,        // String index 2
                                     geh_serail_number,  // String index 3
                                     geh_config          // String index 4
                                     geh_interface       // String index 5
                                     };

#define USB_FILL_CFG_DESC(size, nifc, cfg_id, str_ndx, attrib, pow) \
 (hcc_u8)0x09u, STDD_CONFIG, (hcc_u8)(size), (hcc_u8)((size) >> 8)\
 , (hcc_u8)(nifc), (hcc_u8)(cfg_id), (hcc_u8)(str_ndx), (hcc_u8)(attrib),
   (hcc_u8)(pow)

#define USB_FILL_IFC_DESC(ifc_id, alt_set, no_ep, iclass, isubclass, iproto, strndx) \
 (hcc_u8)0x09u, STDD_INTERFACE,(hcc_u8)(ifc_id), (hcc_u8)(alt_set), (hcc_u8)(no_ep)\
 , (hcc_u8)(iclass), (hcc_u8)(isubclass), (hcc_u8)(iproto), (hcc_u8)(strndx)

#define USB_FILL_EP_DESC(addr, dir, attrib, psize, interval) \
 (hcc_u8)0x07u, STDD_ENDPOINT, (hcc_u8)((addr)&0x7f) | (((hcc_u8)(dir))<<0x7)\
 , (hcc_u8)(attrib), (hcc_u8)((psize) & 0xff), (hcc_u8)(((psize) >> 8) & 0xff)\
 , (interval)

hcc_u8 geh_config_descriptor[] = {
USB_FILL_CFG_DESC(
       9+3+9+9+7,     // Descriptor has 37 bytes
       1,             // 1 configuration
       1,             // This is configuration number 1
       4,             //  String index 4 describes this configuration
CFGD_ATTR_SELF_PWR,  //  Device is self powered
       0),            //  Devices requires 0*2ma of current

USB_FILL_OTG_DESC(
       1,             // OTG disabled
       1),

USB_FILL_IFC_DESC(
GEH_IFC_INDEX,// Interface ID = 0
       0,             // Alternate setting = 0
       1,             // Number of endpoints excluding endpoint 0
       0x3,           // HID interface class
       0x0,           // subclass 0
       0x0,           // Protocol 0
       5),            // String index 5 describes this interface
```

```
USB_FILL_HID_DESC(9, 0x0100, 0x0, 1, 0x22, sizeof(geh_report_descriptor)),

USB_FILL_EP_DESC(
      0x1,           // Endpoint 1
      1,             // IN
      0x3,           // Interrupt transfers
      8,             // Packet Size = 8 bytes
      0x20),         // Host data polling interval = 32ms
```

7 The USB Device-Side Driver

The low-level USB driver interfaces directly with the USB OTG module. There are two separate drivers, one for host and another for device. The file usb.c and its header file usb.h are the device-side USB low-level drivers. On the host side, the files usb_host.c, and usb_host.h contain the low-level driver.

The USB device driver is in the file usb.c. The driver provides the low-level functions for initializing the USB OTG module, and transmitting data. The heart of the driver is the ISR. The USB OTG module has seven interrupt sources. All the interrupt sources are ORed together to create a single interrupt vector.

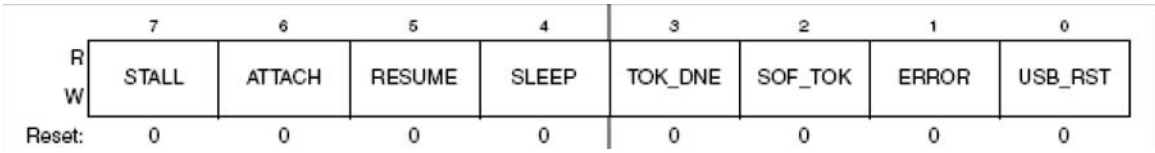


Figure 22. USB Module Interrupt Sources

Table 15. Bit Descriptions

Field	Description
7 STALL	Stall Interrupt In target mode this bit is asserted when a STALL handshake is sent by the SIE. This interrupt can be use to determine if the last USB transaction was completed successfully or if it stalled.
6 ATTACH	Attach Interrupt This bit is set when the USB module detects the attach of a USB device. This signal is only valid if HOST_MODE_EN is true. This interrupt signifies that a peripheral is now present and must be configured.
5 RESUME	This bit is set depending upon the DP/DM signals, and can be used to signal remote wake-up signaling on the USB bus. When not in suspend mode, this interrupt should be disabled.
4 SLEEP	This bit is set when the USB module detects a constant idle on the USB bus for three milliseconds. The sleep timer is reset by activity on the USB bus.
3 TOK_DNE	This bit is set when the current token being processed has completed. The ColdFire core should immediately read the STAT register to determine the endpoint and BD used for this token. Clearing this bit (by writing a one) causes the STAT register to be cleared or the STAT holding register to be loaded into the STAT register.
2 SOF_TOK	This bit is set when the USB module receives a SOF token. In host mode this bit is set when the SOF threshold is reached, so that software can prepare for the next SOF.
1 ERROR	This bit is set when any of the error conditions within the ERR_STAT register occur. The ColdFire core must then read the ERR_STAT register to determine the source of the error.

Table 15. Bit Descriptions (continued)

Field	Description
0 USB_RST	This bit is set when the USB module has decoded a valid USB reset. This informs the microprocessor that it should write 0x00 into the address register and enable Endpoint 0. USB_RST is set after a USB reset has been detected for 2.5 microseconds. It is not be asserted again until the USB reset condition has been removed and then reasserted.

The TOK_DNE interrupt occurs after a the current token is processed. This occurs in either a TX or RX. The driver must read the status register to determine whether the last transaction was a TX or RX, and to determine which buffer descriptor was used.

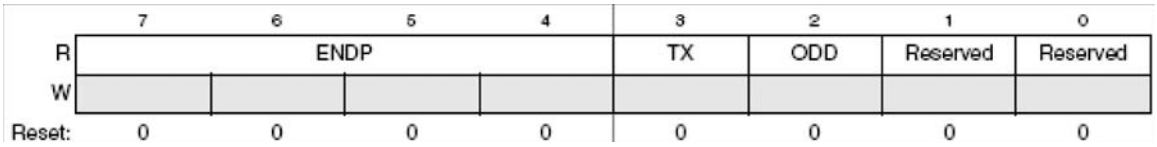


Figure 23. USB Module Status Register

The ENDP field is the endpoint number the last transaction occurred on, the TX bit determines the direction (0 = RX, 1 = TX), and the ODD field determine which of the two buffer descriptors was used. The driver uses this data plus the BDT_PAGE register data to calculate the buffer descriptor absolute address.

```
#define BDT_CTL_RX(ep, b)     (BDT_BASE[((ep)<<3)+((b)<<1)+0])
#define BDT_ADR_RX(ep, b)     (BDT_BASE[((ep)<<3)+((b)<<1)+1])
#define BDT_CTL_TX(ep, b)     (BDT_BASE[((ep)<<3)+((b)<<1)+4])
#define BDT_ADR_TX(ep, b)     (BDT_BASE[((ep)<<3)+((b)<<1)+5])
```

Where ep is the ENDP field, the endpoint number. The b parameter is the ODD field.

The OWN bit is cleared by the driver to indicate that the buffer descriptor is owned by the firmware.

The driver interfaces with the next level above the stack using the ep_info data structure. There is one ep_info structure for each endpoint. These structures are stored in a array declared in usb.c.

```
static ep_info_t ep_info[16];

typedef struct {
                    volatile hcc_u32 tlength;
                    volatile hcc_u32 maxlength;
                    void * volatile address;
                    volatile usb_callback_t data_func;
                    hcc_u16 psize;
                    hcc_u32 data0_tx;
                    hcc_u32 data0_rx;
                    volatile hcc_u8 state;
                    volatile hcc_u8 flags;
                    volatile hcc_u8 error;
                    hcc_u8 next_rx;
                    hcc_u8 next_tx;
              } ep_info_t;
```

Table 16. ep_info Data Structure

Field	Description
tlength	Transfer length (the number of bytes to transfer, decremented after each packet is transferred)
maxlength	Maximum length of transfer
address	Pointer to data buffer
data_func	If: • The buffer is empty and more data needs to be sent • All transmission is finished • In case of an error
psize	Maximum packet size allowed for endpoint
data0_tx	tx toggle bit
data0_rx	rx toggle bit
state	Control endpoint state machine state values EPST_IDLE EPST_DATA_TX EPST_DATA_TX_LAST EPST_DATA_RX EPST_STATUS_TX EPST_STATUS_RX EPST_TX_STOP EPST_ABORT_TX EPST_DATA_TX_WAIT_DB EPST_DATA_TX_EMPTY_DB
flags	Endpoint flag bits EPFL_ERROR // There was an error during the ongoing transfer. EPFL_ZPACKET // After the last data packet an additional zero length packet needs to be transmitted to close the transfer.
error	Error flags: USBEPERR_NONE // No error USBEPERR_TO_MANY_DATA // To many data received USBEPERR_PROTOCOL_ERROR // Protocol error USBEPERR_USER_ABORT // Transfer was aborted by the application USBEPERR_HOST_ABORT // Host aborted the transfer
next_rx	Next buffer to be used. The USB module supports two buffers / endpoint direction. This specifies the next buffer to use.
next_tx	Next buffer to be used. The USB module supports two buffers / endpoint direction. This specifies the next buffer to use.

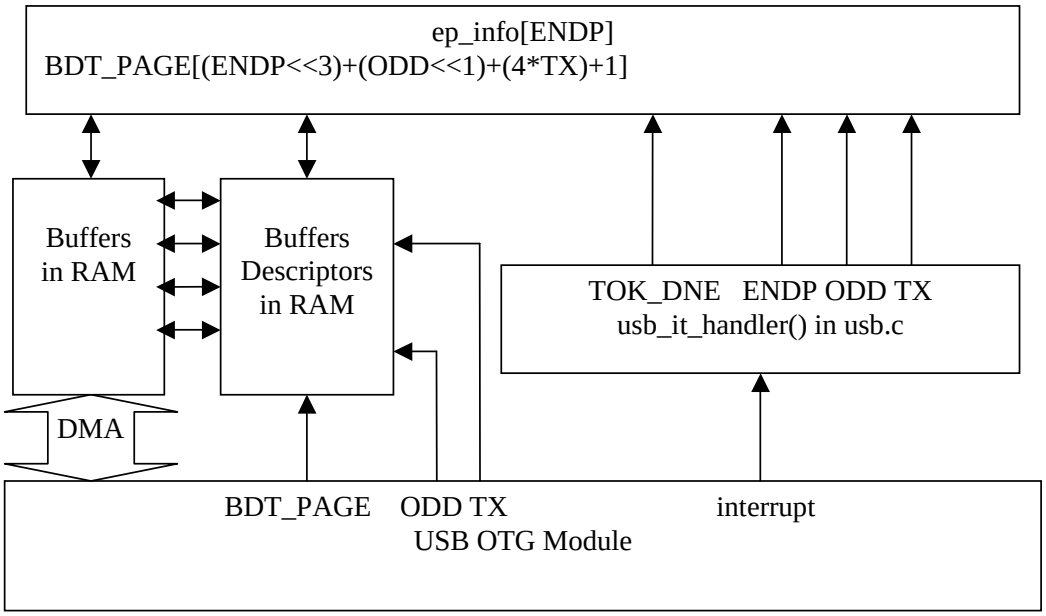


Figure 24. USB Device-Side Driver Block Diagram

7.1 Usb_it_handler()

The heart of the USB driver is the interrupt handler. The USB module generates interrupts from eight different events as specified by the USB module specification.

Table 17. USB Module Interrupt Sources

Interrupt	usb_it_handler() Response
USB_RST	The handler resets its state machine, disables all endpoints, and resets Endpoint 0. usb_state is set to USBST_DEFAULT. The usb_reset_event() callback function is called to indicate the reset to the higher layers of the stack.
ERROR	The usb_bus_error_event() callback function is called.
RESUME	The usb_wakeup_event() callback function is called.
SLEEP	The usb_suspend_event() callback function is called.
STALL	The control endpoint is reset.
SOF_TOK	This interrupt is not used by usb_it_handler().
ATTACH	This interrupt is not used by usb_it_handler().
TOK_DNE	All transactions start with a token packet sent by the host. In device mode, this interrupt occurs after the device receives the token packet from the host. This is where the communication takes place. This portion of the interrupt processes packets for the IN (TX) endpoints, the OUT (RX) endpoint, and control endpoints.

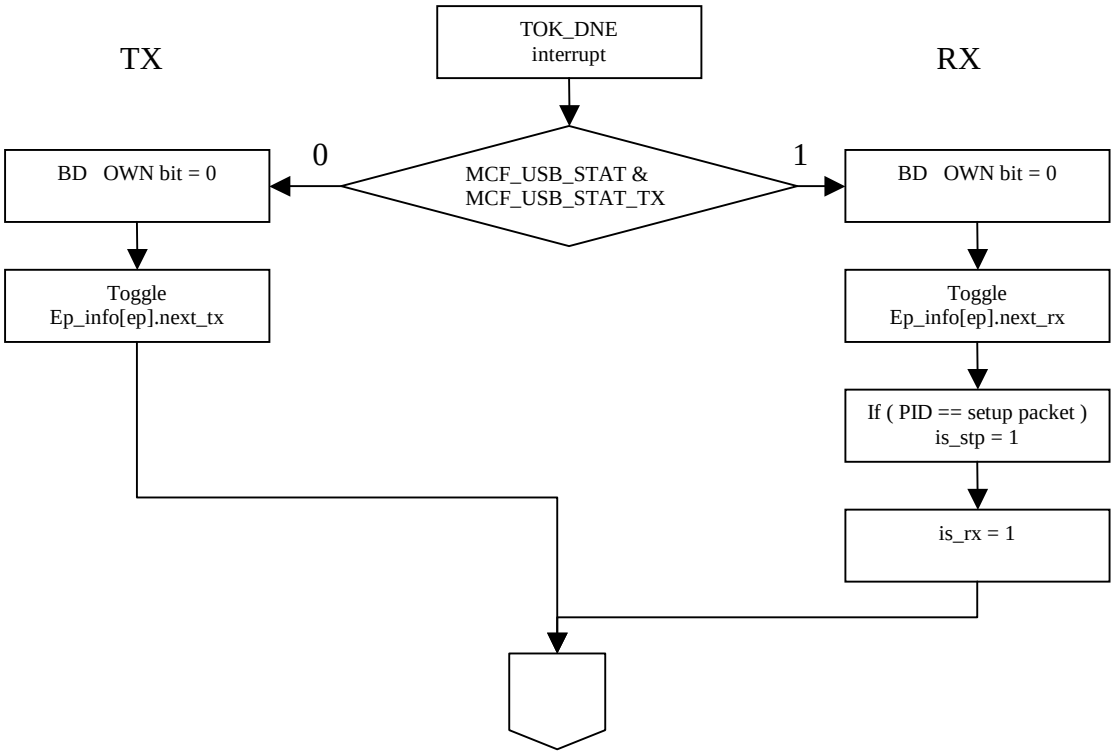


Figure 25. TOK_DNE Interrupt Flow

After the interrupt has claimed the buffer descriptors by clearing the OWN bit, and the buffer is toggled in the ep_info structure. If this packet is a RX packet, then the PID is checked to determine if it is a SETUP packet. The is_stp flag is set if it is a setup packet. Setup packets for control endpoints are handled by a state machine.

The MCF_USB_ENDPT_EP_CTL_DIS flag is located in the endpoint control registers. This bit, when set, disables control (SETUP) transfers. When cleared, control transfers are enabled. If the packet received is from a endpoint supporting control transfers, then the MCF_USB_ENDPT_EP_CTL_DIS flag is cleared, allowing SETUP packets to be received on the endpoint. Control endpoints are supported by a state machine. Control transfers contain multiple tokens: a setup token indicates the start of the transfer, an in or out token indicates the start of data, and a handshake token for handshaking. The state machine is entered after each of these tokens. The state machine starts in the IDLE_STATE. While in the IDLE_STATE, it is waiting for a setup packet to indicate the start of a control transfer.

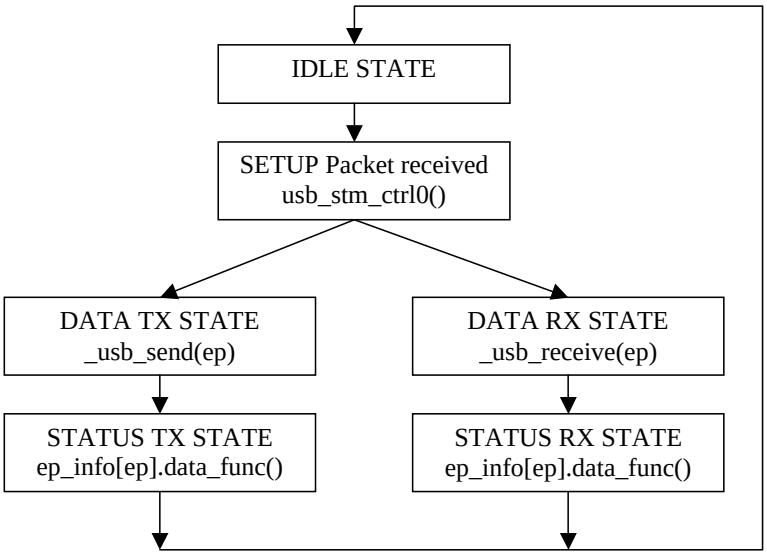


Figure 26. Control Endpoint State Machine

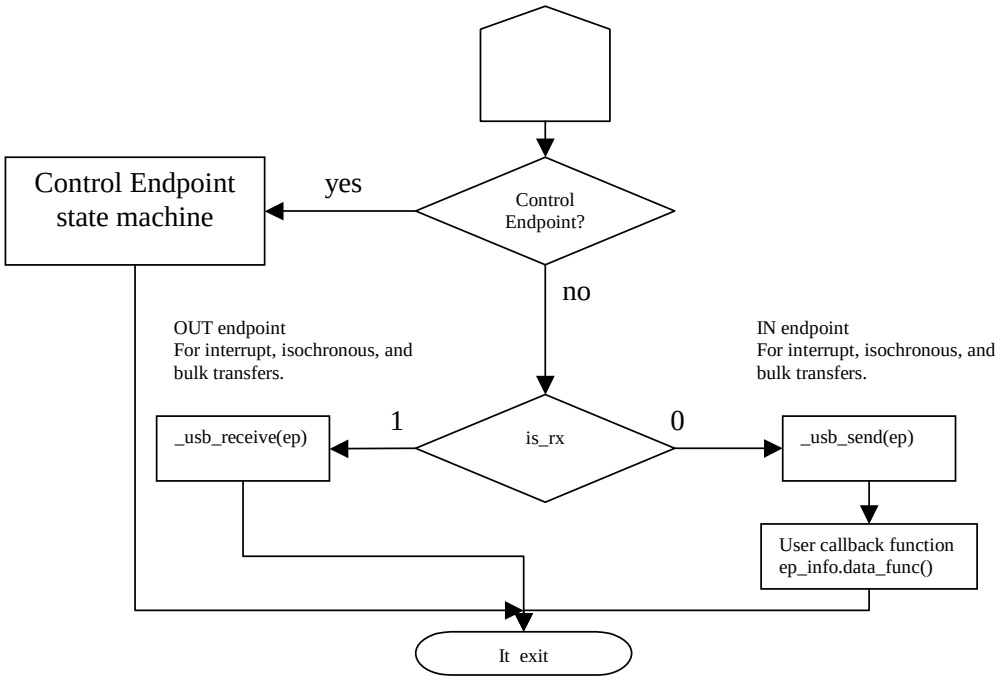


Figure 27. In, Out, and Control Endpoint Processing

7.2 Enumeration Support—usb_stm_ctrl0()

The setup packet contains 8 bytes of data. This data is used to support the 11 standard USB requests required for enumeration. These requests are parsed in the usb_stm_ctrl0() function. The usb_stm_ctrl0() function is called by the USB interrupt handler when a SETUP packet is received. The 8 bytes contain the request, and parameters for the request. Requests are always sent from the host.

USBRQ_SET_ADDRESS

This request sets the device address for all future device accesses.
new_address = device address sent from host

The function cb_set_address() is called at the end of this transaction.

USBRQ_GET_DESCRIPTOR

This request returns the specified descriptor if the descriptor exists.

The host specifies the descriptor it is asking for. The device sends the descriptor to the host during the data phase.

USBRQ_GET_CONFIGURATION

This request returns the current device configuration value.

If the returned value is zero, the device is not configured.

USBRQ_SET_CONFIGURATION

This request sets the device configuration.

The host specifies the configuration that the device should use.

The firmware calls set_config() with the configuration number sent by the host.

USBRQ_CLEAR_FEATURE

This request is used to clear or disable a specific feature.

The firmware disables the endpoint specified by the host.

7.3 USB Device-Side API

```
hcc_u8 usb_init(hcc_u8 ip, hcc_u8 use_alt_clk)

       The usb_init() function is used to initialize the usb driver.
       ip = interrupt level assigned to the USB module
       us_alt_clk = Clock source select for module

void usb_stop(void)
       Disable USB interrupts, and shutdown the USB module.

void usb_send      hcc_u8 ep, usb_callback_t f, hcc_u8* data, hcc_u32 tr_length,
                   hcc_u32 req_length)
       Setup a TX (IN) transfer.  The data will be transferred the next time the host
       request data from the endpoint. Because all packet transmission on USB are started
       by the host, it needs to know how many bytes shall be transferred during a
       transfer. Because of this the host will always tell the device how many bytes it can
       receive (req_length). On the other hand, the the device may have less data ready
       (tr_length).  Calls the _usb_send( ep ) function.
```

```
       ep = endpoint number
       f  = pointer to user callback function. A callback will be made
              if:  - the buffer is empty and more data needs to be sent
                   - all transmission is finished
                   - in case of an error
       data = pointer to data buffer
       buffer_size = size of data buffer
       tr_length   = number of bytes to be transferred.
       req_length  = the number of bytes the host wants to receive.

void usb_receive(    hcc_u8 ep, usb_callback_t f, hcc_u8* data, hcc_u32 tr_length,
                     hcc_u32 req_length)
       Setup a RX (OUT) transfer on the endpoint.  When the host sends data the
       callback function is called.  The data can be found in the ep_info structure.
       Parameters are same as usb_send().

hcc_u8 usb_ep_is_busy(hcc_u8 ep)
       returns nonzero if endpoint is busy (a transfer is ongoing).

hcc_u8 usb_get_state(void)
       returns USB state

hcc_u8 usb_ep_error(hcc_u8 ep)
       Returns endpoint ( ep ) specific error code.
```

7.4 USB Driver Callback Functions

```
callback_state_t usb_ep0_callback(void)
       This callback function is called when a SETUP packet is received on a
       control endpoint.  The USB driver has already processed any USB standard
       packets for enumeration.  This callback is used to implement class specific
       requests.

void usb_bus_error_event(void)
       USB callback function. Is called by the USB driver if an USB error event
       occurs.

void usb_wakeup_event(void)
       USB callback function. Is called by the USB driver if an USB wakeup event
       occurs.

void usb_suspend_event(void)
       USB callback function. Is called by the USB driver if an USB suspend event
       occurs.

void usb_reset_event(void)
       USB callback function. Is called by the USB driver if an USB reset event
       occurs.
```

8 The USB Host-Side Driver

The USB host-side driver is in the files usb_host.c and usb_utils.c. The host-side driver is simple thanks to the advanced USB hardware. The host-side driver does not require interrupts. Host mode allows bulk, isochronous, interrupt, and control transfers. Bulk data transfers are performed at nearly the full USB bus bandwidth. Support is provided for ISO transfers, but the number of ISO streams that can be practically supported is affected by the interrupt latency of the processor servicing the token during interrupts from the SIE.

8.1 Initializing the Host Controller (host_init() in usb_host.c)

1. Setup the buffer table and point to first RX / TX buffer descriptors

1 ODD_RST	Setting this bit to 1 resets all the BDT ODD ping/pong bits to 0, which then specifies the EVEN BDT bank.

```
// use first rx and tx buffer
MCF_USB_CTL |= MCF_USB_CTL_ODD_RST;
```

Field	Description
7 – 1 BDT_ BA[15:8]	This 7 bit field provides address bits 15 through 9 of the BDT base address, which defines where the Buffer Descriptor Table resides in system memory.
0 NOT USED	This bit is always zero. The 32-bit BDT Base Address is always aligned on 512 byte boundaries in memory.

```
// set BDT address
MCF_USB_BDT_PAGE_01 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 8);
MCF_USB_BDT_PAGE_02 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 16);
MCF_USB_BDT_PAGE_03 = (hcc_u8)(((hcc_u32)BDT_BASE) >> 24);
```

2. Configure the SOF threshold

Field	Description
7 – 0 CNT[7:0]	This 8 bit field represents the SOF count threshold in byte times.

```
MCF_USB_SOF_THLDL = 74;   // disable transactions 74 bytes before SOF
```

3. Enable pulldown resistors

5 DP_LOW	D+ Data Line pull-down resistor enable 0 D+ pull-down resistor is not enabled 1 D+ pull-down resistor is enabled This bit should always be enabled together with bit 4 (DM_LOW)
4 DM_LOW	D- Data Line pull-down resistor enable 0 D- pull-down resistor is not enabled 1 D- pull-down resistor is enabled This bit should always be enabled together with bit 5 (DP_LOW)
2 OTG_EN	On-The-Go pull-up/pull-down resistor enable 0 If USB_EN is set and HOST_MODE is clear in the Control Register (CTL), then the D+ Data Line pull-up resistors are enabled. If HOST_MODE is set the D+ and D- Data Line pull-down resistors are engaged. 1 The pull-up and pull-down controls in this register are used

```
MCF_USB_OTG_CTRL = MCF_USB_OTG_CTRL_DP_LOW | MCF_USB_OTG_CTRL_DM_LOW |
                   MCF_USB_OTG_CTRL_OTG_EN;
MCF_GPIO_PQSPAR |= MCF_GPIO_PQSPAR_PQSPAR5(3) | MCF_GPIO_PQSPAR_PQSPAR6(3);
```

4. Enable host mode (CTL[HOST_MODE_EN]=1). Pulldown resistors enabled, pullup disabled. SOF generation begins. SOF counter loaded with 12,000. Eliminate noise on the USB by disabling start-of-frame packet generation by writing the USB enable bit to 0 (CTL[USB_EN]=0).

0 USB_EN/ SOF_EN	USB Enable 0 The USB Module is disabled 1 The USB Module is enabled. Setting this bit causes the SIE to reset all of its ODD bits to the BDTs. Therefore, setting this bit resets much of the logic in the SIE. When host mode is enabled, clearing this bit causes the SIE to stop sending SOF tokens.

```
MCF_USB_CTL = MCF_USB_CTL_HOST_MODE_EN;
```

5. Enable the ATTACH interrupt (INT_ENB[ATTACH]=1) or clear the ATTACH flag if polling (to clear the flag, write a 1).

6 ATTACH	Attach Interrupt This bit is set when the USB Module detects an attach of a USB device. This signal is only valid if HOST_MODE_EN is true. This interrupt signifies that a peripheral is now present and must be configured.

```
MCF_USB_INT_STAT = MCF_USB_INT_STAT_ATTACH;
```

6. Wait for ATTACH interrupt (INT_STAT[ATTACH]) or poll. Signaled by USB Target pull-up resistor changing the state of DPLUS or DMINUS from 0 to 1 (SE0 to J or K state).

```
if (MCF_USB_INT_STAT & MCF_USB_INT_STAT_ATTACH) device is attached
```

7. (optional) Check the state of the JSTATE and SE0 bits in the control register. If the JSTATE bit is 0, then the connecting device is low speed. If the connecting device is low speed then set the low-speed bit in the address registers (ADDR[LS_EN]=1) and the host the host without hub bit in Endpoint 0 register control (EP_CTL0[HOST_WO_HUB]=1).

7 JSTATE	Live USB differential receiver JSTATE signal. The polarity of this signal is affected by the current state of LS_EN (See)
6 SE0	Live USB Single Ended Zero signal

8. Enable RESET (CTL[RESET]=1) for 10 ms.

4 RESET	Setting this bit enables the USB Module to generate USB reset signaling. This allows the USB Module to reset USB peripherals. This control signal is only valid in Host mode (HOST_MODE_EN=1). Software must set RESET to 1 for the required amount of time and then clear it to 0 to end reset signaling. For more information on RESET signaling see Section 7.1.4.3 of the USB specification version 1.0.

```
MCF_USB_CTL |= MCF_USB_CTL_RESET;            // start reset signaling Delay 10ms
MCF_USB_CTL &= ~MCF_USB_CTL_RESET;           // stop reset signaling
MCF_USB_INT_STAT = MCF_USB_INT_STAT_USB_RST; // clear reset event
```

9. Enable the SOF packet to keep the connected device from going to suspend (CTL[USB_EN=1]).

0 USB_EN/ SOF_EN	USB Enable 0 The USB Module is disabled 1 The USB Module is enabled. Setting this bit causes the SIE to reset all of its ODD bits to the BDTs. Therefore, setting this bit resets much of the logic in the SIE. When host mode is enabled, clearing this bit causes the SIE to stop sending SOF tokens.

```
MCF_USB_CTL |= MCF_USB_CTL_USB_EN_SOF_EN;
```

10. Set up the endpoint control register for bidirectional control transfers.
    `EP_CTL0[4:0] = 0x0d.`

8.2 usb_host_transaction() in usb_host.c

With USB, data is transferred in transactions. The hosts initiates all transactions by sending a token to the device. There are three tokens: in, out, and setup. Setup is a special token used to start a control transfer. The driver only supports control transfers on Endpoint 0. This is not a limitation of the USB specification, but a limitation of the free USB stack.

The usb_host_transaction() function generates one transaction of the type specified in the command line. For control transfers, the usb_host_function() will be called three times to support the token, data, and handshaking phases.

```
static hcc_u16 usb_host_start_transaction(hcc_u8 type, hcc_u8 *buffer, hcc_u16 length,
hcc_u8 ep)

     where: type = type of transaction to initiate ( define in usb_host.h )
                   TRT_SETUP = SETUP transaction
                   TRT_IN    = IN transaction
                   TRT_OUT   = OUT transaction

            *buffer = Pointer to data area for transfer
            length  = Size of the buffer in bytes
            ep      = EndPoint handle
```

The ep parameter requires a little more explanation. The structure my_device, declared in usb_host.c, is used by the host firmware to encapsulate the device information. There is only one my_device entry. The example host firmware does not support hubs; therefore, only one device is supported.

```
dev_table_element_t my_device;          // declared in usb_host.c

typedef struct
{
  hcc_u8 address;                       // Device address
  hcc_u8 low_speed;                     // Low Speed device flag
  device_ep_t eps[MAX_EP_PER_DEVICE];   // Device endpoints
} dev_table_element_t;
```

```
typedef struct
{
  hcc_u16 last_due;                    // Interval due time
  hcc_u16 psize;                       // Packet size
  hcc_u8 type;                         // Endpoint type
  hcc_u8 address;                      // Endpoint Address
  hcc_u8 interval;                     // Interrupt frame interval
  hcc_u8 tgl_rx;                       // RX DATA toggle state
  hcc_u8 tgl_tx;                       // TX DATA toggle state
} device_ep_t;
```

8.3 usb_host_transaction() Pseudo Code

```
static hcc_u16 usb_host_start_transaction(hcc_u8 type, hcc_u8 *buffer, hcc_u16 length,
hcc_u8 ep)
{
```

The device address is set using the ADDR register. If the device is a low-speed device, the LS_EN flag is set to force the USB OTG module to create a preamble.

7 LS_EN	Low Speed Enable bit. This bit informs the USB Module that the next token command written to the token register must be performed at low speed. This enables the USB Module to perform the necessary preamble required for low-speed data transmissions.
6–0 ADDR	USB address. This 7-bit value defines the USB address that the USB Module decodes in peripheral mode, or transmit when in host mode.

```
/* Set device address. */
MCF_USB_ADDR = (hcc_u8)(my_device.low_speed ?
               my_device.address | MCF_USB_ADDR_LS_EN :
               my_device.address);
```

The host firmware does not support a hub.

Field	Description
7 HOST_WO _HUB	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit allows the host to communicate to a directly connected low speed device. When cleared, the host produces the PRE_PID then switch to low speed signaling when sending a token to a low speed device as required to communicate with a low speed device through a hub.
6 RETRY_DIS	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit causes the host to not retry NAK'ed (Negative Acknowledgement) transactions. When a transaction is NAKed, the BDT PID field is updated with the NAK PID, and the TOKEN_DNE interrupt is set. When this bit is cleared NAKed transactions is retried in hardware. This bit must be set when the host is attempting to poll an interrupt endpoint.

```
/* We are not talking over a HUB. */
MCF_USB_ENDPT0 = MCF_USB_ENDPT0_HOST_WO_HUB;
```

Interrupt endpoints are limited to only one transaction per *x* frames, where *x* is the interval defined for the endpoint. The endpoint interval is defined in my_device.eps[ep].interval. The last_due variable contains the frame number for when it is safe to send the next interrupt transaction. Because frames occur every millisecond, frame numbers translate to time in milliseconds. Therefore, the code below waits for interval ms after the last transaction, before sending a transaction.

```
/* If this is an interrupt endpoint wait till the endpoint is due. */
if (my_device.eps[ep].type == EPTYPE_INT)
{
hcc_u16 elapsed;
```

6 RETRY_DIS	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit causes the host to not retry NAK'ed (Negative Acknowledgement) transactions. When a transaction is NAKed, the BDT PID field is updated with the NAK PID, and the TOKEN_DNE interrupt is set. When this bit is cleared NAKed transactions is retried in hardware. This bit must be set when the host is attempting to poll an interrupt endpoint.

```
/* Disable hardware retry. */
MCF_USB_ENDPT0 |= MCF_USB_ENDPT0_RETRY_DIS;
/* Disable software retry. */
retry=1;

/* wait till frame is due */
do {
elapsed=(hcc_u16)(MCF_USB_FRM_NUM-my_device.eps[ep].last_due);
elapsed &= ((1<<11)-1);
} while(elapsed < my_device.eps[ep].interval);

Increment last_due by interval for next transaction

my_device.eps[ep].last_due += my_device.eps[ep].interval;
my_device.eps[ep].last_due &= ((1<<11)-1);
}
```

The following switch statement will do a setup, in, or out transaction depending on the type of transaction requested by the function call (type is a input parameter to this function).

The do loop is a software retry loop. If the transaction completes in a error, the software will attempt retries.

```
do  {
        tr_error=tre_none;
        retry--;
        switch(type)
        {
```

8.4 usb_host_transaction()—Setup Transaction

A setup transaction consists of a setup packet, followed by data from the host, with the device sending a handshake packet. A setup packet is part of a control transfer. The driver forces Endpoint 0 for control transfers as a limitation of the free USB stack.

```
case TRT_SETUP:
       /* Configure bidirectional communication. */
       /* Usb spec says setup packets shall be accepted
          even if the device was not able to process its packet
          buffer. Thus NAK handshake should not be given to a
          setup packet by the device. Anyway some devices will
          happily NAK setup packets :( */
```

RETRY_DIS is only set for a interrupt packet (see above). Retries are enabled here to support non-compliant devices that NAK setup transactions. Control endpoints are bidirectional, so both the RX and TX are enabled. Handshaking is enabled.

6 RETRY_DIS	This is a Host mode only bit and is only present in the control register for endpoint 0 (ENDPT0). When set this bit causes the host to not retry NAK'ed (Negative Acknowledgement) transactions. When a transaction is NAKed, the BDT PID field is updated with the NAK PID, and the TOKEN_DNE interrupt is set. When this bit is cleared NAKed transactions is retried in hardware. This bit must be set when the host is attempting to poll an interrupt endpoint.
4 EP_CTL_DIS	This bit, when set, disables control (SETUP) transfers. When cleared, control transfers are enabled. This applies if and only if the EP_RX_EN and EP_TX_EN bits are also set. SeeTable 16-34
3 EP_RX_EN	This bit, when set, enables the endpoint for RX transfers. SeeTable 16-34
2 EP_TX_EN	This bit, when set, enables the endpoint for TX transfers. SeeTable 16-34
1 EP_STALL	When set this bit indicates that the endpoint is called. This bit has priority over all other control bits in the EndPoint Enable Register, but it is only valid if EP_TX_EN=1 or EP_RX_EN=1. Any access to this endpoint causes the USB Module to return a STALL handshake. After an endpoint is stalled it requires intervention from the Host Controller.
0 EP_HSHK	When set this bet enables an endpoint to perform handshaking during a transaction to this endpoint. This bit is generally set unless the endpoint is Isochronous.

```
MCF_USB_ENDPT0 |=  0x0d;
```

For a setup transaction, the data toggles are reset to 1.

```
/* After the setup we shall send/receive DATA1 packets. */
my_device.eps[ep].tgl_tx=1;
my_device.eps[ep].tgl_rx=1;
```

During a setup transaction, the host sends a data packet. To do this, you need to setup the buffer descriptor appropriately. The OWN bit is set to indicate that the USB OTG module owns the buffer descriptor, and can send the data when appropriate. There is no actual data transmission at this time, the transmission is being setup to happen automatically after the TOKEN packet is transmitted.

The BDT_ADR_TX macro generates a pointer to a buffer descriptor address field. A pointer to buffer is inserted into the address field.

```
/* Set data buffer address. */
WR_LE32(&BDT_ADR_TX(0, ep_info.next_tx), (hcc_u32)buffer);
```

The BDT_CTL_TX macro generates a pointer to a buffer descriptor control word. The byte count (BC) field of the control word is set to eight (8 bytes of data transfer). The OWN bit is set.

31:26	25:16	15:8	7	6	5	4	3	2	1	0
RSVD	BC (10-Bits)	RSVD	OWN	DATA0/1	KEEP/ TOK_PID[3]	NINC/ TOK_PID[2]	DTS/ 5 TOK_PID[1]	BDT_STALL/ TOK_PID[0]	0	0
Buffer Address (32-Bits)										

```
/* Set packet properties and give buffer to USB. */
bdt_ctl=&BDT_CTL_TX(0, ep_info.next_tx);
WR_LE32(bdt_ctl, (0x8<<16) | BDT_CTL_OWN | 0);
```

At this point, you must wait until the last transaction is complete. This is done by polling the TOKBUSY bit.

5 TXSUSPEND/ TOKENBUSY	When the USB Module is in Host mode TOKEN_BUSY is set when the USB Module is busy executing a USB token and no more token commands should be written to the Token Register. Software should check this bit before writing any tokens to the Token Register to ensure that token commands are not lost. In Target mode TXD_SUSPEND is set when the SIE has disabled packet transmission and reception. Clearing this bit allows the SIE to continue token processing. This bit is set by the SIE when a Setup Token is received allowing software to dequeue any pending packet transactions in the BDT before resuming token processing.

```
/* Wait until pending tokens are processed. */
while(MCF_USB_CTL & MCF_USB_CTL_TXDSUSPEND_TOKBUSY)
{
}
```

At this point, begin the transaction by writing to the TOKEN register.
TOKEN_SETUP(PID) = 0x0D.

	7	6	5	4	3	2	1	0
R W	TOKEN_PID				TOKEN_ENDPT			
Reset:	0	0	0	0	0	0	0	0

```
/* Start transaction. */
MCF_USB_TOKEN=(hcc_u8)((TOKEN_SETUP<<4) | (my_device.eps[ep].address | (0<<7)));
break;
```

8.5 usb_host_transaction()—In Transaction

```
    case TRT_IN:
           /* Configure bi-directional communication + auto repeat. */
           MCF_USB_ENDPT0 |= 0x0d;

           /* Set RX buffer address. */
           WR_LE32(&BDT_ADR_RX(0, ep_info.next_rx), (hcc_u32)buffer);

           /* Set packet properties and give next buffer to USB. */
           bdt_ctl=&BDT_CTL_RX(0, ep_info.next_rx);
           WR_LE32(bdt_ctl, (length<<16) | BDT_CTL_OWN |
                        my_device.eps[ep].tgl_rx);
           my_device.eps[ep].tgl_rx =(hcc_u8)(my_device.eps[ep].tgl_rx ? 0 :
                        BDT_CTL_DATA);

           /* Wait till pending tokens are processed. */
           while(MCF_USB_CTL & MCF_USB_CTL_TXDSUSPEND_TOKBUSY)
           {
           }

           /* Start transaction. */
           MCF_USB_TOKEN=(hcc_u8)((TOKEN_IN<<4) |
                        (my_device.eps[ep].address | (1<<7)));
           reak;
```

8.6 usb_host_transaction()—Out Transaction

```
    case TRT_OUT:
           /* Configure bi-directional communication + auto repeat. */
           MCF_USB_ENDPT0 |= 0x0d;

           /* Set TX buffer address. */
           WR_LE32(&BDT_ADR_TX(0, ep_info.next_tx), (hcc_u32)buffer);

           /* Set packet properties and give buffer to USB. */
           bdt_ctl=&BDT_CTL_TX(0, ep_info.next_tx);
           WR_LE32(bdt_ctl, (length<<16) | BDT_CTL_OWN |
                       my_device.eps[ep].tgl_tx);
           my_device.eps[ep].tgl_tx = (hcc_u8)(my_device.eps[ep].tgl_tx ? 0 :
                       BDT_CTL_DATA);

           /* Wait till pending tokens are processed. */
           while(MCF_USB_CTL & MCF_USB_CTL_TXDSUSPEND_TOKBUSY)
           {
           }

           /* Start transaction. */
           MCF_USB_TOKEN=(hcc_u8)((TOKEN_OUT<<4) |
                        (my_device.eps[ep].address | (0<<7)));
           break;
// End of switch(type)
```

8.7 usb_host_transaction()—Transaction Complete

After the transaction is initiated above, wait for the entire transaction to complete by polling the TOK_DNE bit.

3 TOK_DNE	This bit is set when the current token being processed has completed. The ColdFire core should immediately read the STAT register to determine the EndPoint and BD used for this token. Clearing this bit (by writing a one) causes the STAT register to be cleared or the STAT holding register to be loaded into the STAT register.

```
/* Wait for transaction end. */
while((MCF_USB_INT_STAT & (MCF_USB_INT_STAT_TOK_DNE | MCF_USB_INT_STAT_STALL |
MCF_USB_INT_STAT_ERROR)) ==0)
{
}
```

Check for errors:

```
    if ((MCF_USB_ERR_STAT & ~MCF_USB_ERR_STAT_CRC5_EOF) != 0)
{
MCF_USB_ERR_STAT = 0xff;
MCF_USB_INT_STAT = MCF_USB_INT_STAT_ERROR;
tr_error=tre_data_error;
continue;
}
```

Acknowledge the TOK_DNE signal:

```
/* device not disconnected while waiting for an aswer */
/* clear transfer ok event */
MCF_USB_INT_STAT=MCF_USB_INT_STAT_TOK_DNE;
MCF_USB_CTL &= ~MCF_USB_CTL_TXDSUSPEND_TOKBUSY;
```

8.8 usb_host_transaction()—Data Toggling

USB uses a one-bit sequence number, referred to as a data toggle. After transmitting or receiving data, we toggle the toggle bit for the next transaction.

```
/* switch to next buffer */
if (type== TRT_SETUP || type == TRT_OUT)
{
  ep_info.next_tx ^= 0x1u;
}
else
{
  ep_info.next_rx ^= 0x1u;
}
```

8.9 usb_host_transaction()—Transaction Status

The status of the last transaction is returned in the TOK_PID field of the buffer descriptor. The token variable below is assigned the 4 bits in the TOK_PID field.

TOK_PID[*n*]	Bits [5:2] can also represent the current token PID. The current token PID is written back in to the BD by the USB-FS when a transfer completes. The values written back are the token PID values from the USB specification: 0x1 for an OUT token, 0x9 for and IN token or 0xd for a SETUP token. In host mode this field is used to report the last returned PID or a transfer status indication. The possible values returned are: 0x3 DATA0, 0xb DATA1, 0x2 ACK, 0xe STALL, 0xa NAK, 0x0 Bus Timeout, 0xf Data Error.

```
/* Check transaction status.*/
token=(hcc_u8)((RD_LE32(bdt_ctl) >> 2) & 0x0f);
```

token = TOK_PID for buffer descriptor from last transaction. TOK_PID contains the transfer status.

```
    /* transaction accepted by device */
    switch (token)
    {
         default:
         case TOKEN_ACK:  // TOKEN_ACK = 0x02 defined in usb_host.c
                return((hcc_u16)((RD_LE32(bdt_ctl) >> 16u) & 0x3ffu));

         case TOKEN_NAK: // TOKEN_NAK = 0x0A defined in usb_host.c
                 /* device is not ready */
                if (my_device.eps[ep].type == EPTYPE_INT)
                {
                        return(0);
                }
                /* retry */
                break;        // by not returning,  we go around the loop again

         case TOKEN_STALL: /* endpoint stalled by device */
                /* we can not get here because we already checked
                 INT_STAT_STALL. */
                tr_error=tre_stall;
                return((hcc_u16)-1u);

         case 0xf:  /* data error, retry */
                tr_error=tre_data_error;
                break;

         case 0:  /* no answer, retry. */
                tr_error=tre_silent;
                break;
    }
    } while(retry);

    return((hcc_u16)-1u);
}
```

8.10 USB Host-Side Driver API

8.10.1 USB Host-Side Driver API—Sending / Receiving Data

These four functions are used to send data to the device, or receive data from the device. These functions are essentially wrappers around the usb_host_start_transaction() function. These functions block until the transfer is complete or there is an error.

8.10.1.1 hcc_u16 host_send_control(hcc_u8 *setup_data, hcc_u8* buffer, hcc_u8 ep)

setup_data = pointer to 8 bytes of data to send in SETUP phase.

buffer = Pointer to data buffer to be sent during data phase.

ep = handle to endpoint (index into my_device.eps[ep])

The host_send_control() function performs an out control transfer. Control transfers have three phases: setup, data, handshake. This function creates all three phases. Sending the data pointed to by buffer during the DATA phase.

The SETUP data size is fixed at 8 bytes.

If the number of bytes to send is > packet size, then multiple DATA transactions are performed. The ep parameter (endpoint handle) is passed directly to the usb_host_start_transaction() function. The ep parameter is an index into the eps array of the my_device structure. This function is used to send requests and or data to the device.

The function returns –1 on error, or number of transmitted bytes

Pseudo code:

```
hcc_u16 host_send_control(hcc_u8 *setup_data, hcc_u8* buffer, hcc_u8 ep)
{
      hcc_u32 curr=0;
      hcc_u16 length=RD_LE16(setup_data+6);

      /* setup transaction. */
       usb_host_start_transaction(TRT_SETUP, setup_data, 8, ep);

      /* data transactions */
      while(curr<length)
      {
           hcc_u16 psize=(hcc_u16)(MIN(my_device.eps[ep].psize, length));
          hcc_u8 r=(hcc_u8)usb_host_start_transaction(TRT_OUT, buffer+curr, psize, ep);
           if (r != psize)
                    return((hcc_u16)-1u);

      curr += psize;
      }

      /* handshake transaction */
      my_device.eps[ep].tgl_rx = BDT_CTL_DATA;
      usb_host_start_transaction(TRT_IN, (void *)0, 0, ep);
```

```
        return((hcc_u16)curr);
}
```

8.10.1.2 hcc_u32 host_send(hcc_u8* buffer, hcc_u32 length, hcc_u8 ep)

buffer = pointer to data to send.

length = # of bytes to send.

ep = handle to endpoint (index into my_device.eps[ep])

The host_send() function creates a OUT transfer for non-control endpoints. This function is used to send data to the device. OUT transfers consists of three packets: TOKEN, DATA, followed by a handshake packet from the device. The usb_host_start_transaction() function does most of the work.

Returns –1 on error, or number of bytes sent.

Pseudo code:

```
hcc_u32 host_send(hcc_u8* buffer, hcc_u32 length, hcc_u8 ep)
{
        hcc_u32 curr=0;

        /* data transactions */
        while(curr<length)
        {
                hcc_u16 psize=(hcc_u16)(MIN(my_device.eps[ep].psize, length));
                /* do transaction */
                usb_host_start_transaction(TRT_OUT, buffer+curr, psize, ep);
                curr += psize;
        }

        return(curr);
}
```

8.10.1.3 hcc_u32 host_receive(hcc_u8* buffer, hcc_u32 length, hcc_u8 ep)

buffer = pointer to storage area for received data.

length = size of buffer.

ep = handle to endpoint (index into my_device.eps[ep])

The host_receive() function performs a IN transfer on a non-control endpoint. This function is used to receive data from a device.This function is similar to host_send().

Pseudo code:

```
hcc_u32 host_receive(hcc_u8* buffer, hcc_u32 length, hcc_u8 ep)
{
        hcc_u32 curr=0;

        /* data transactions */
        while(curr<length)
        {
                hcc_u16 got,
```

```
                psize=(hcc_u16)(MIN(my_device.eps[ep].psize, length));
                got=usb_host_start_transaction(TRT_IN, buffer+curr, psize, ep);
                if (got == ((hcc_u16)-1u))
                      break;

                curr += got;

                /* short packet means end of transfer */
                if (got != my_device.eps[ep].psize)
                      break;
        }

        return(curr);
}
```

8.10.1.4 hcc_u16 host_receive_control(hcc_u8 *setup_data, hcc_u8* buffer, hcc_u8 ep)

The host_receive_control() function performs a IN transfer on a control endpoint. This function is used to receive data from a device on a control endpoint. For instance, during enumeration, this function is used to receive a descriptor.

The DATA packet in the SETUP transaction is limited to carrying eight bytes (all USB standard requests are eight bytes). Also, there is no data length in the parameter list because this function assumes a USB standard request packet is being sent in the SETUP stage; therefore, the size of the data transfer is determined from byte 6 in the setup data packet (see USB standard requests in the enumeration section).

```
setup_data = pointer to 8 bytes of data to send in SETUP phase.
buffer = pointer to storage area for received data.
ep = handle to endpoint ( index into my_device.eps[ep] )

pseudo code

hcc_u16 host_receive_control(hcc_u8 *setup_data, hcc_u8* buffer, hcc_u8 ep)
{
        hcc_u32 curr=0;
        hcc_u16 length = RD_LE16(setup_data+6);

        /* setup transaction. */
        usb_host_start_transaction(TRT_SETUP, setup_data, 8, ep);

        /* data transactions */
        while(curr<length)
        {
                hcc_u16 got,
                psize=(hcc_u16)(MIN(my_device.eps[ep].psize, length));
                got=usb_host_start_transaction(TRT_IN, buffer+curr, psize, ep);
                curr += got;
                if (got == ((hcc_u16)-1u))
                        return((hcc_u16)-1u);

                /* short packet means end of transfer */
                if (got != my_device.eps[ep].psize)
                        break;
        }
```

```
        /* handshake transaction */
        my_device.eps[0].tgl_tx = BDT_CTL_DATA;
        if (((hcc_u16)-1u)==usb_host_start_transaction(TRT_OUT, (void *)0, 0, ep))
             return((hcc_u16)-1u);

    return((hcc_u16)curr);
     }
```

8.10.2 USB Host-Side Driver API—Endpoint Management

Each device connected to the host has an entry in the my_device array. Because the host firmware is limited to accepting one device only, there is only one entry in the array. Each device entry in the my_device array includes a eps array. The eps array is used to describe a endpoint. When a data transfer function is called with a endpoint handle, the handle is an index into the eps array.

```
EndPoint Structure ( EPS )
      hcc_u16 last_due;     // Set during transaction
      hcc_u16 psize;        // Set by functions below
      hcc_u8 type;          // Set by functions below
      hcc_u8 address;       // Set by functions below
      hcc_u8 interval;      // Set by functions below
      hcc_u8 tgl_rx;        // Set during transaction
      hcc_u8 tgl_tx;        // Set during transaction
```

There are three functions used to manage the eps array: host_add_ep(), host_remove_ep(), and host_modify_ep().

8.10.2.1 hcc_u8 host_add_ep(hcc_u8 type, hcc_u8 address, hcc_u8 interval, hcc_u16 psize)

type = type of endpoint (defined in usb_host.h) -> eps.type

EPTYPE_CTRL

EPTYPE_ISO

EPTYPE_BULK

EPTYPE_INT

address = address of endpoint = 0 to 0x0f -> eps.address

interval = poll interval for interrupt endpoints -> eps.interval

psize = maximum packet size for endpoint -> eps.psize

The host_add_ep() function adds an endpoint into the eps array for the device supported. The function returns a handle that is actually an index into the eps array. The handle is not the endpoint address. If a –1 is returned, then there are no more endpoints available for the device.

8.10.2.2 void host_remove_ep(hcc_u8 ep_handle)

The host_remove_ep() function sets the eps[ep_handle].address = INVALID_ADDRESS, marking the endpoint as empty.

8.10.2.3 void host_modify_ep(hcc_u8 ep_handle, hcc_u8 type, hcc_u8 address, hcc_u8 interval, hcc_u16 psize)

ep_handle = handle returned from host_add_ep().

type = type of endpoint (defined in usb_host.h) -> eps[ep_handle].type

EPTYPE_CTRL

EPTYPE_ISO

EPTYPE_BULK

EPTYPE_INT

address = address of endpoint = 0 to 0x0f -> eps[ep_handle].address

interval = poll interval for interrupt endpoints -> eps[ep_handle].interval

psize = maximum packet size for endpoint -> eps[ep_handle].psize

Modifies eps[ep_handle]

8.10.3 USB Host-Side Driver API—Device Management

8.10.3.1 hcc_u8 host_has_device(void)

The host_has_device() function calls the chk_dev() function. Returns a true if(my_device.address != INVALID_ADDRESS). This is a indication that the device remains connected.

```
void host_reset_bus(void)
```

Set the USB_CTL_RESET for 10 ms to perform reset signaling on the USB bus. If a device is detected, configure Endpoint 0 to get device descriptor/Endpoint 0 packet size. Then set device address to 1.

8.10.3.2 void host_init(void)

See detailed description in Section 8.1, “Initializing the Host Controller (host_init() in usb_host.c).”

8.10.3.3 int host_scan_for_device(void)

Calls chk_dev() to see if device is connected, and if so, calls host_reset_bus(). Returns 1 if a device is connected and host received device descriptor.

8.10.3.4 void host_stop(void)

Shuts down the USB OTG module.

8.10.3.5 void host_sleep(void)

Disables SOF transmissions.

8.10.3.6 void host_wakeup(void)

Enables SOF transmissions.

8.11 usb_utils.c

The following functions are used during the process of enumeration.

8.11.1 void fill_setup_packet(hcc_u8* dst, hcc_u8 dir, hcc_u8 type, hcc_u8 recipient, hcc_u8 req, hcc_u16 val, hcc_u16 ndx, hcc_u16 len)

This function is used to fill out a standard USB request setup data packet.

```
dst = a pointer to 8 byte buffer
dir = Data Direction – macros defined in usb_utils.h
STP_DIR_IN
STP_DIR_OUT
type = Request type
STP_TYPE_STD
STP_TYPE_CLASS
STP_TYPE_VENDOR
recipient =  What is requested
STP_RECIPIENT_DEVICE
STP_RECIPIENT_IFC
STP_RECIPIENT_ENDPT
STP_RECIPIENT_OTHER
req = Request ID ( either standard USB requests, or HID requests )
STDRQ_GET_STATUS                ( usb_utils.h )
STDRQ_CLEAR_FEATURE             ( usb_utils.h )
STDRQ_SET_FEATURE               ( usb_utils.h )
STDRQ_SET_ADDRESS               ( usb_utils.h )
STDRQ_GET_DESCRIPTOR            ( usb_utils.h )
STDRQ_SET_DESCRIPTOR            ( usb_utils.h )
STDRQ_GET_CONFIGURATION         ( usb_utils.h )
STDRQ_SET_CONFIGURATION         ( usb_utils.h )
STDRQ_GET_INTERFACE             ( usb_utils.h )
STDRQ_SET_INTERFACE             ( usb_utils.h )
STDRQ_SYNCH_FRAME               ( usb_utils.h )
HIDRQ_GET_REPORT                ( host_hid.c )
HIDRQ_GET_IDLE                  ( host_hid.c )
HIDRQ_GET_PROTOCOL              ( host_hid.c )
HIDRQ_SET_REPORT                ( host_hid.c )
HIDRQ_SET_IDLE                  ( host_hid.c )
HIDRQ_SET_PROTOCOL              ( host_hid.c )
val = Request specific Data
STDDTYPE_DEVICE                 ( usb_utils.h )
STDDTYPE_CONFIGURATION          ( usb_utils.h )
STDDTYPE_STRING                 ( usb_utils.h )
STDDTYPE_INTERFACE              ( usb_utils.h )
STDDTYPE_ENDPOINT               ( usb_utils.h )
ndx =   index or offset
len = Number of bytes to transfer
```

Table 18. void fill_setup_packet Field Descriptions

Offset	Field/ Parameter	Size	Value	Description
0	bmRequestType / (dir|type|recipient)	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-drive 1 = Device-to-host D6 . .. 5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4 ... 0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4 ... 31 = Reserved
1	bRequest / req	1	Request #	Specific request
2	wValue / val	2	Value	Word-sized field that varies according to request
4	wIndex / ndx	2	Index or Offset	Index or Offset Word-sized field that varies according to request typically used to pass an index or offset
6	wLength / len	2	Count	Number of bytes to transfer if int get_dev_desc(void)

8.11.2 int get_dev_desc(void)

The get_dev_desc() function requests a device descriptor from the device. The device descriptor is stored in dbuffer. dbuffer is a global data area of 255 bytes (declared in usb_utils.c). A setup data packet is generated using the fill_setup_packet() function, then sent using the host_receive_control() function. Returns 0 on success.

```
fill_setup_packet(
               setup,                   // 8 byte array to store setup packet
               STP_DIR_IN,              // Data from device
               STP_TYPE_STD,            // USB standard request
               STP_RECIPIENT_DEVICE,    // Request for device
               STDRQ_GET_DESCRIPTOR,    // Get descriptor
               (STDDTYPE_DEVICE<<8)|0,  // descriptor type is device
               0,                       // NA for this request
               18);                     // Expect 18 bytes of data

host_receive_control( setup, dbuffer, 0 );
```

8.11.3 int get_cfg_desc (hcc_u8 ndx)

The get_cfg_desc() function requests a configuration descriptor from the device. The configuration descriptor can be any size. The host has no idea how big the descriptor is until the descriptor is received. The get_cfg_desc() function first sends a configuration request with a length of 5. Bytes 2 and 3 are the total length (word) of the configuration descriptor. After receiving the partial configuration descriptor, a request is made for the complete descriptor.

Table 19. Configuration Descriptors

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	CONFIGURATION Descriptor Type (0x02)
2	wTotalLength	2	Number	Total length of data returned for this configuration.

The configuration descriptor is stored in dbuffer. dbuffer is a global data area of 255 bytes (declared in usb_utils.c). A setup data packet is generated using the fill_setup_packet() function, then sent using the host_receive_control() function. Returns 0 on success.

```
    fill_setup_packet(
        setup,                      // 8 byte array to store setup packet
        STP_DIR_IN,                 // Data from device
        STP_TYPE_STD,               // USB standard request
        STP_RECIPIENT_DEVICE,       // Request for device
           STDRQ_GET_DESCRIPTOR,    // Get descriptor
        (STDDTYPE_CONFIGURATION<<8)| // type configuration
        ndx),                       // configuration index
        0,                          // NA for this request
        5);                         // Read first 5 bytes of desc

    host_receive_control( setup, dbuffer, 0 );

    fill_setup_packet(
        setup,                      // 8 byte array to store setup packet
        STP_DIR_IN,                 // Data from device
        STP_TYPE_STD,               // USB standard request
        STP_RECIPIENT_DEVICE,       // Request for device
        STDRQ_GET_DESCRIPTOR,       // Get descriptor
        (STDDTYPE_CONFIGURATION<<8)| // type configuration
        ndx)                        // configuration index
        0,                          // NA for this request
        RD_LE16(dbuffer+2));        // Read complete descriptor

host_receive_control( setup, dbuffer, 0 );
```

8.11.4 int set_ep0_psize(void)

The set_ep0_psize() function performs a read device descriptor operation, then calls host_modify_ep() to configure Endpoint 0 packet size. The device descriptor is stored in dbuffer. This function is used during enumeration to setup Endpoint 0. Returns a 0 on success.

8.11.5 int set_address(hcc_u8 address)

The set_address() function sends a set address request to the device.

```
fill_setup_packet(
    setup,                      // pointer to 8 byte array
    STP_DIR_OUT,                // OUT transfer
    STP_TYPE_STD,               // Standard request
    STP_RECIPIENT_DEVICE,       // Request for device
      STDRQ_SET_ADDRESS,        // set address request
    address,                    // address to set device to
    0,                          // NA
    0);                         // NA

host_send_control(setup, dbuffer, 0);

/* we need to wait maximum 50 mS to let the device change its address. */
   host_ms_delay(45);
```

8.11.6 int set_config(hcc_u8 cfg)

Sets the device configuration using a set configuration request.

```
fill_setup_packet(
    setup,                      // pointer to 8 byte array
    STP_DIR_OUT,                // OUT transfer
    STP_TYPE_STD,               // Standard request
    STP_RECIPIENT_DEVICE,       // request for device
    STDRQ_SET_CONFIGURATION,    // Set configuration
    cfg,                        // config to use
    0,                          // NA
    0);                         // NA

    host_send_control(setup, dbuffer, 0);
```

8.11.7 int get_device_info(device_info_t *res)

Sends a get device descriptor request to the device using the get_dev_desc() function call. The device descriptor is parsed into a device_info_t structure, and stored at the address pointed to by res. Returns a 0 on success.

```
get_dev_desc();       // Reads device descriptor into dbuffer

 /* give read values to caller */
 res->clas=DEVDESC_CLASS(dbuffer);
 res->sclas=DEVDESC_SCLASS(dbuffer);
 res->protocol=DEVDESC_PROTOCOL(dbuffer);
 res->rev=DEVDESC_REV(dbuffer);
 res->vid=DEVDESC_VID(dbuffer);
 res->pid=DEVDESC_PID(dbuffer);
 res->ncfg=DEVDESC_NCFG(dbuffer);
```

8.11.8 int get_cfg_info(cfg_info_t *res)

Parses a configuration descriptor into a cfg_info_t structure. The data is stored at the address pointed to by res. This function does not request a configuration descriptor from the device. It assumes the configuration descriptor is stored in dbuffer.

```
res->nifc=CONFDESC_INTRFACES(dbuffer);
res->ndx=CONFDESC_MY_NDX(dbuffer);
res->str=CONFDESC_MY_STR(dbuffer);
res->attrib=CONFDESC_ATTRIB(dbuffer);
res->max_power=CONFDESC_MAX_POW(dbuffer);
```

8.11.9 int get_ifc_info(ifc_info_t *res, hcc_u16 offset)

Parses the interface descriptor indexed by offset. The parsed interface descriptor is stored in the ifc_info_t structure pointed to by res.

```
res->clas=IFCDESC_CLASS(&dbuffer[offset]);
res->sclas=IFCDESC_SCLASS(&dbuffer[offset]);
res->protocol=IFCDESC_PROTOCOL(&dbuffer[offset]);
res->ndx=IFCDESC_MY_NDX(&dbuffer[offset]);
res->alt_set=IFCDESC_ALTERNATE(&dbuffer[offset]);
res->str=IFCDESC_MY_STR(&dbuffer[offset]);
res->nep=IFCDESC_ENDPONTS(&dbuffer[offset]);
```

8.11.10 int get_ep_info(ep_info_t *res, hcc_u16 offset)

Parses the endpoint descriptor indexed by offset. The parsed endpoint descriptor is stored in the ep_info_t structure pointed to by res.

```
res->address=EPDESC_ADDRESS(&dbuffer[offset]);
res->type=EPDESC_ATTRIB(&dbuffer[offset]);
res->interval=EPDESC_INTERVAL(&dbuffer[offset]);
res->psize=EPDESC_PSIZE(&dbuffer[offset]);
```

8.12 Host Firmware—host_scan_for_device()

The host_scan_for_device() function call performs most of the device enumeration. The majority of the work is done in the host_reset_bus() call and the chk_dev() call.

```
Pseudo code

int host_scan_for_device(void)
{
      if (chk_dev())           // Check for ATTACH indication
      {
           host_reset_bus();    // Start Enumeration
           return(1);
 }

      return(0);
}
Pseudo code

static hcc_u8 chk_dev(void)
```

```
{
      // Verify that device is not already enumerated
      if (my_device.address == INVALID_ADDRESS)
      {
          int x;
          /* If we can not clear the attach flag, then a device is connected. */
          MCF_USB_INT_STAT = MCF_USB_INT_STAT_ATTACH;

          /* Some delay is needed between clearing the ATTACH flag, and checking
          it again. Unfortunately there is no information about the length of
          the delay. */
          for(x=0; x< 10000; x++);

          if (MCF_USB_INT_STAT & MCF_USB_INT_STAT_ATTACH)
          evt_connect();
      }

      return((hcc_u8)(my_device.address != INVALID_ADDRESS));
}
```

The evt_connect() function verifies that device is attached, sets up Endpoint 0, and determines the speed of the device (full or low).

```
static hcc_u8 evt_connect(void)
{
      hcc_u8 ep;
      int x;

      /* debounce (100 mS) */
      host_ms_delay(100);

      /* clear attach event */
      MCF_USB_INT_STAT = MCF_USB_INT_STAT_ATTACH;

      /* Some delay is needed between clearing the ATTACH flag, and checking
      it again. Unfortunately there is no information about the length of
      the delay. */
      for(x=0; x<10000; x++) ;

      /* Is a device connected? */
      if((MCF_USB_INT_STAT & MCF_USB_INT_STAT_ATTACH) == 0)
          return(0); // no

      /* A newly connected device shall have address 0, */
      my_device.address=0;

      /* and only ep0 is working. We assume packet size of
      ep0 is the possible minimum. We will read the real
      value during enumeration. */
      host_modify_ep(0, EPTYPE_CTRL, 0, 0, MIN_EP0_PSIZE);

      /* remove all endpoints except 0 */
      for(ep=1;ep<MAX_EP_PER_DEVICE;ep++)
          host_remove_ep(ep);

      /* clear low speed bit to make JSTATE detection consistent. */
      MCF_USB_ADDR=0;
```

```
	/* let settle D+ and D- to the right state */
	host_ms_delay(1);

	/* Check if device is low or high speed. */
	if ((MCF_USB_CTL & MCF_USB_CTL_JSTATE) == 0)
	{
	    /* Low speed device. */
	    my_device.low_speed=1;
	    MCF_USB_ADDR = MCF_USB_ADDR_LS_EN;
	}
	else
	    my_device.low_speed=0;

	return(1);
}
void host_reset_bus(void)
{
	hcc_u8 ep=0;

	/********* reset */
	/* Start reset signaling. */
	MCF_USB_CTL |= MCF_USB_CTL_RESET;

	/* The minimum reset signal length is 10 mS. We use an 1 mS timer so it has
	at least 1 mS error (1 period). USB specifies +-0.05% accuracy for frame
	interval. So we are good enough if we wait for 11 SOF due times. */
	host_ms_delay(11);

	/* stop reset signaling */
	MCF_USB_CTL &= ~MCF_USB_CTL_RESET;

	/* Clear reset event. */
	MCF_USB_INT_STAT = MCF_USB_INT_STAT_USB_RST;

	/********* do firmware reset */
	/* A reset device shall have address 0, */
	my_device.address=0;

	/* and only ep0 is working. We assume packet size of
	ep0 is the possible minimum. We will read the real
	value during enumeration. */
	host_modify_ep(0, EPTYPE_CTRL, 0, 0, MIN_EP0_PSIZE);

	/* remove all endpoints except 0 */
	for(ep=1;ep<MAX_EP_PER_DEVICE;ep++)
	            host_remove_ep(ep);

	/* check if we have a device connected */
	/* if a device is connected, it will answer to address 0 */
	if (chk_dev())
	{
	            /* enable SOF generation */
	            MCF_USB_CTL |= MCF_USB_CTL_USB_EN_SOF_EN;
	            MCF_USB_INT_STAT = MCF_USB_INT_STAT_SLEEP |
	                               MCF_USB_INT_STAT_RESUME;

	            /* device may need max 10 mS reset recovery time */
```

```
                host_ms_delay(100);

                /* Read out endpoint 0 packet size. */
                if (set_ep0_psize())
                        return;

                /* set device address */
                if (set_address(1))
                        return;

                my_device.address=1;
        }
}
```

8.13 Host Firmware—Device Enumeration

After the USB OTG module is initialized via a call to host_init(), the host firmware continuously calls host_scan_for_device().

Host_init():

1. The device is plugged into a host. The host provides power to the device with a current limit of 100 ma. The host looks for a device attach indication.
2. The host determines low-speed/full-speed capability by pullup resistors connected to either the D+ or D– lines. At this point, the device is in the powered state.
3. The host sends a reset to the device, by setting D+ and D– low for at least 10 milliseconds. When the host removes the reset, the device goes into the default state.
4. In the default state, the device is ready to respond to control transfers at Endpoint 0. The host communicates with the device using the default address of 00. The device can draw up to 100 ma from the host.
5. The host sends a GET_DESCRIPTOR request to Endpoint 0, address 0, to learn the maximum packet size of the default pipe. The eight bytes of the device descriptor contains the maximum packet size supported by Endpoint 0.
6. The host assigns a unique address to the device by sending a SET_ADDRESS request. The device is now in the address state.
   while(!host_scan_for_device());
7. The host sends a GET_DESCRIPTOR request to the new address to read the full device descriptor.
   get_dev_desc(); // The device descriptor is in dbuffer
8. The host then requests any additional descriptors specified in the device descriptor. Each descriptor begins its length and type.
   get_cfg_desc(1); // The configuration descriptor is in dbuffer
9. The host assigns a device driver based on the data in the descriptors. Windows will use the devices vendor ID and product ID to find an appropriate INF file to determine what drivers to load. If there is no match, Windows uses a default driver according to class.
10. If the device supports multiple configurations, the host sends a SET_CONFIGURATION request to the device to select the desired configuration.
    set_config(1);

8.14 emg_host_demo()

The following code is an example of how to enumerate a device using the host API. In this example, the firmware prints out the device and configuration descriptors to the serial port (38400, 8, n, 1). A new function (not part of the standard stack) was written to request the string descriptors from the device.

```
//
// Enumerate device, and output device / configuration descriptors to the serial port in hex
// Serial descriptors are also printed to the serial port
//
// Written by Eric Gregori (847) 651 - 1971
//
int main(void)
{
      hcc_u8                      cfg, str1, str2, str3;
      hcc_u16                     length, i;
      device_info_t   dev_inf;
      cfg_info_t                  cfg_inf;


      hw_init();
      uart_init(38400, 1, 'n', 8);
      host_init();

      print( "\r\nHost Demo by Eric Gregori\n\r" );
      print("EMG Host application started.\r\n");

      while(1)
      {
          busy_wait();

          /* a device is already connected, wait till it is disconnected */
          print("Waiting for device removal.\r\n");
          while(host_has_device());       // Spin waiting for !ATTACH

          print("Device disconnected.\r\n");

          /* At this point no device is attached. Wait till attachment. */
          print("Waiting for device...\r\n");
          while(!host_scan_for_device());  // Spin waiting for ATTACH

          print("Device connected.\r\n");

          // Read and parse device descriptor
          // get_device_info() calls get_dev_desc()
          if( !get_device_info(&dev_inf) )
          {
                 print( "\n\rDevice Descriptor\n\r" );
                 for( i=0; i<18; i++ )
                 {
                      emg_printbytehex( dbuffer[i] );
                      print( " " );
                 }
                 print( "\n\r" );

                 str1 = dbuffer[14];
                 str2 = dbuffer[15];
```

```
                str3 = dbuffer[16];

                if( str1 )
                {
                     print( "\r\nManufacture: " );
                     emg_print_str_desc( str1 );
                }

                if( str2 )
                {
                     print( "\r\nProduct: " );
                     emg_print_str_desc( str2 );
                }

                if( str3 )
                {
                     print( "\r\nSerial Number: " );
                     emg_print_str_desc( str3 );
                }

                print( "\r\n\r\nDecoded Device Descriptor" );
                print( "\n\ridVendor = " );
                emg_printwordhex( dev_inf.vid );
                print( "\n\ridProduct = " );
                emg_printwordhex( dev_inf.pid );
                print( "\n\rbcdDevice = " );
                emg_printwordhex( dev_inf.rev );
                print( "\n\rbDeviceClass = " );
                emg_printbytehex( dev_inf.clas );
                print( "\n\rbDeviceSubClass = " );
                emg_printbytehex( dev_inf.sclas );
                print( "\n\rbDeviceProtocol = " );
                emg_printbytehex( dev_inf.protocol );
                print( "\n\rbNumConfigurations = " );
                emg_printbytehex( dev_inf.ncfg );
                print( "\n\r" );
}
else
      print( "\r\nFailure Reading Device Descriptor" );

      // Read all configuration descriptors
      for(cfg=0; cfg < dev_inf.ncfg; cfg++)
      {
      // get the configuration descriptor
      if (get_cfg_desc(cfg))
                continue;                     // Descriptor cfg not found
      else
      {
      print( "\n\rConfiguration Descriptor - " );
      emg_printbytehex( cfg );
      length=RD_LE16(dbuffer+2);
                for( i=0; i<length; i++ )
                {
                       if( (i%16) == 0 )
                                print( "\n\r" );
                       emg_printbytehex( dbuffer[i] );
                       print( " " );
```

```
                }

                str1 = dbuffer[6];

                print( "\n\r\n\rDecoded Configuration Descriptor" );

                // Call get_cfg_info() to parse configuration descriptor
                get_cfg_info( &cfg_inf );

          print( "\n\rbNumInterfaces = " );
                emg_printbytehex( cfg_inf.nifc );
          print( "\n\rbConfigurationValue = " );
                emg_printbytehex( cfg_inf.ndx );
          print( "\n\riConfiguration = " );
                emg_printbytehex( cfg_inf.str );
          print( "\n\rbmAttributes = " );
                emg_printbytehex( cfg_inf.attrib );
          print( "\n\rbMaxPower * 2ma = " );
                emg_printbytehex( cfg_inf.max_power );

                if( str1 )
                {
                      print( "\r\nManufacture: " );
                      emg_print_str_desc( str1 );
                }
          }
          print( "\n\r\n\r" );
     } // end of config descriptor read

} //
// while(1)
```

8.15 Displaying a String Descriptor—emg_print_str_desc()

```
void emg_print_str_desc( unsigned char desc )
{
      unsigned chari;

      if( !emg_get_str_descriptor( desc ) )
      {
           // Unicoded string is in dbuffer starting at 2
           // strlen = (dbuffer[0] - 2)*2
           for( i=2; i<=(dbuffer[0]-2); i+=2 )
                uart_putch( dbuffer[i] );
      }
}
```

8.16 emg_get_str_descriptor()

```
int emg_get_str_descriptor( unsigned char desc )
{
      hcc_u8        setup[8];
      hcc_u16       length=3;
      hcc_u8        retry=3;

      std_error=stderr_none;
      do
      {
          // Build SETUP data packet
          fill_setup_packet(setup, STP_DIR_IN, STP_TYPE_STD, STP_RECIPIENT_DEVICE,
          STDRQ_GET_DESCRIPTOR, (hcc_u16)((STDDTYPE_STRING<<8)|desc), 0, length);
          if (length == host_receive_control(setup, dbuffer, 0))
          {
                 /* Check returned descriptor type and length (ignore extra bytes) */
                 if ((USBDSC_TYPE(dbuffer) == STDDTYPE_STRING))
                 {
                        length=dbuffer[0];

                        if( length >= DBUFFER_SIZE )
                               length = DBUFFER_SIZE-1;

                        // Rebuild SETUP data packet with new length
                        fill_setup_packet(setup, STP_DIR_IN, STP_TYPE_STD,
                               STP_RECIPIENT_DEVICE,
                               STDRQ_GET_DESCRIPTOR,
                               (hcc_u16)((STDDTYPE_STRING<<8)|desc),
                               0, length);

                        if (length == host_receive_control(setup, dbuffer, 0))
                               return(0);
                 }
          }
      }while(retry--);

      std_error=stderr_host;
      return(1);
}
```

8.17 emg_host_demo()—Camera 1 Enumeration

```
      Host Demo by Eric Gregori
      EMG Host application started.
      Waiting for device removal.
      Device disconnected.
      Waiting for device...
      Device connected.

      Device Descriptor
      12 01 10 01 FF FF FF 08 45 05 33 83 01 00 00 00 00 01


      Decoded Device Descriptor
      idVendor = 0545
      idProduct = 8333
```

```
bcdDevice = 0001
bDeviceClass = FF
bDeviceSubClass = FF
bDeviceProtocol = FF
bNumConfigurations = 01

Configuration Descriptor - 00
09 02 89 00 01 01 00 80 32 09 04 00 00 01 FF FF
FF 00 07 05 81 01 00 00 01 09 04 00 01 01 FF FF
FF 00 07 05 81 01 00 01 01 09 04 00 02 01 FF FF
FF 00 07 05 81 01 80 01 01 09 04 00 03 01 FF FF
FF 00 07 05 81 01 00 02 01 09 04 00 04 01 FF FF
FF 00 07 05 81 01 80 02 01 09 04 00 05 01 FF FF
FF 00 07 05 81 01 00 03 01 09 04 00 06 01 FF FF
FF 00 07 05 81 01 80 03 01 09 04 00 07 01 FF FF
FF 00 07 05 81 01 FF 03 01

Decoded Configuration Descriptor
bNumInterfaces = 01
bConfigurationValue = 01
iConfiguration = 00
bmAttributes = 80
bMaxPower * 2ma = 32

Waiting for device removal.
```

8.18 emg_host_demo()—Camera 2 Enumeration

```
Device disconnected.
Waiting for device...
Device connected.

Device Descriptor
12 01 10 01 00 00 00 40 45 0C 0D 60 01 01 00 01 00 01

Product:

Decoded Device Descriptor
idVendor = 0C45
idProduct = 600D
bcdDevice = 0101
bDeviceClass = 00
bDeviceSubClass = 00
bDeviceProtocol = 00
bNumConfigurations = 01

Configuration Descriptor - 00
09 02 17 01 01 01 00 80 FA 09 04 00 00 03 FF FF
FF 00 07 05 81 01 00 00 01 07 05 82 02 40 00 00
07 05 83 03 01 00 64 09 04 00 01 03 FF FF FF 00
07 05 81 01 80 00 01 07 05 82 02 40 00 00 07 05
83 03 01 00 64 09 04 00 02 03 FF FF FF 00 07 05
81 01 00 01 01 07 05 82 02 40 00 00 07 05 83 03
01 00 64 09 04 00 03 03 FF FF FF 00 07 05 81 01
80 01 01 07 05 82 02 40 00 00 07 05 83 03 01 00
64 09 04 00 04 03 FF FF FF 00 07 05 81 01 00 02
01 07 05 82 02 40 00 00 07 05 83 03 01 00 64 09
```

```
04 00 05 03 FF FF FF 00 07 05 81 01 A8 02 01 07
05 82 02 40 00 00 07 05 83 03 01 00 64 09 04 00
06 03 FF FF FF 00 07 05 81 01 20 03 01 07 05 82
02 40 00 00 07 05 83 03 01 00 64 09 04 00 07 03
FF FF FF 00 07 05 81 01 84 03 01 07 05 82 02 40
00 00 07 05 83 03 01 00 64 09 04 00 08 03 FF FF
FF 00 07 05 81 01 FF 03 01 07 05 82 02 40 00 00
07 05 83 03 01 00 64

Decoded Configuration Descriptor
bNumInterfaces = 01
bConfigurationValue = 01
iConfiguration = 00
bmAttributes = 80
bMaxPower * 2ma = FA

Waiting for device removal.
```

8.19 emg_host_demo()—Mouse Enumeration

```
Device disconnected.
Waiting for device...
Device connected.

Device Descriptor
12 01 00 02 00 00 00 08 6D 04 16 C0 40 03 01 02 00 01

Manufacture: Logitech
Product: Optical USB Mouse

Decoded Device Descriptor
idVendor = 046D
idProduct = C016
bcdDevice = 0340
bDeviceClass = 00
bDeviceSubClass = 00
bDeviceProtocol = 00
bNumConfigurations = 01

Configuration Descriptor - 00
09 02 22 00 01 01 00 A0 32 09 04 00 00 01 03 01
02 00 09 21 10 01 00 01 22 34 00 07 05 81 03 04
00 0A

Decoded Configuration Descriptor
bNumInterfaces = 01
bConfigurationValue = 01
iConfiguration = 00
bmAttributes = A0
bMaxPower * 2ma = 32

Waiting for device removal.
```

9 The HID Class

One of the first USB classes to be adopted by the Windows operating system was the Human Interface Device (HID) class. Originally part of Windows 98, the HID class has been supported in every version of Microsoft Windows since. The HID class has been supported in Linux since version 2.2.7 of the kernel. Although, it is generally considered unstable in Linux versions before kernel 2.4.

The HID class was designed to provide communications between a input or simple output device and a computer. Example HID devices include: mouses, keyboards, joysticks, front panels, and remote controls. Data transfers are bidirectional and acknowledged.

Table 20. HID Speeds and Limitations

Specification	Bytes/Transaction	Bytes / Second
USB 1.1 Low-speed USB 2.0 Low-speed	8	800
USB 1.1 Full-speed USB 2.0 Full-speed	64	64999

The HID class supports both interrupt and control transfers. If interrupt transfers are used, then the maximum performance number mentioned above can be achieved. If control transfers are used, then the maximum performance may be achieved but not guaranteed. The HID specification can be found at http://www.usb.org/developers/hidpage/. At the time of publication, the specification is Device Class Definition for HID 1.11.

All HID transfers use either the default control pipe or an interrupt pipe for communications to the host. The specification requires that a HID device must have an interrupt in endpoint for sending data to the host. A interrupt out endpoint to receive data from the host is optional.

Table 21. Data and Transfer Types

Transfer Type	Source of Data	Type of Data
Control	Device (IN transfer)	Data that does not have critical timing requirements
Control	Device (OUT transfer)	Data that does not have critical timing requirements, or any data if there is no OUT interrupt pipe.
Interrupt	Device (IN transfer)	Periodic data or data that must be transferred at max rate.
Interrupt	Device (OUT transfer) (optional)	Periodic data or data that must be transferred at max rate.

A application can also use the default control pipe for data transfers. The control pipe for a HID is used for standard USB requests as well as six class-specific requests. The HID specific requests used for data transfers are Set_Report and Get_Report. The Set_Idle, Get_Idle, Set_Protocol, and Get_Protocol are the other four HID requests sent through the default control pipe.

Table 22. HID Specific Requests

Request #	Request	Data Source	Data Length	Data Contents	Required
0x01	Get_Report	Device	Report length	Report	Yes
0x02	Get_Idle	Device	1	Idle duration	No
0x03	Get_Protocol	Device	1	Protocol	Required for boot devices
0x09	Set_Report	Host	Report length	Report	No
0x0A	Set_Idle	Host	0	None	No
0x0B	Set_Protocol	Host	o	None	Required for boot devices

The HID stack uses a report structure to communicate with the application. During enumeration the device shares report descriptors with the host. The report descriptors describe the data that will be transmitted/received in the report structures. There are specifications for these report structures and descriptors, but they tend to be application specific. If the user controls both sides of the communication, and is not expecting to communicate with a standard driver on the PC side (keyboard, mouse, ..) then a generic report structure/descriptor can be used.

Table 23. HID Report Specifications

Specification/File	Usage
Hut1_2.pdf	Mouse, keyboard, joystick, simulation controls, telephone controls, digitizers, bar code scanners, scales, point of sale, and arcade and camera controls
Pid1_01.pdf	Physical interface devices (force feedback joysticks, steering wheels, etc.)
Pdcv10.pdf	Power devices such as UPS
Usbmon10.pdf	Monitor control
Oaaddataformatsv6.pdf	Arcade products (coin mechanisms, bill validators, input pads, and general-purpose I/O)
Pos1_02.pdf	Point of sale devices

9.1 HID Device Firmware

The HID device firmware is located in the file hid.c. The HID stack is initialized with a call to HID_init(). The application has to call the function hid_process() within its main loop to process HID reports through the queuing mechanism. Report entries are created with calls to the hid_add_report() function. After a report entry is created, data is sent with the hid_write_report() call, and received with the hid_read_report() call.

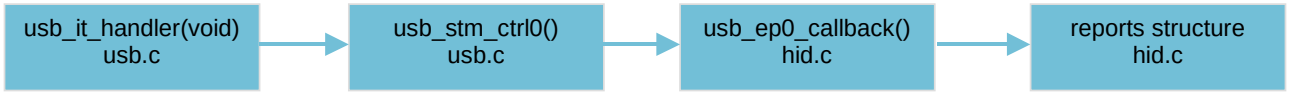
The heart of the HID firmware is the usb_ep0_hid_callback(void) callback function. This function is called by the USB driver Endpoint 0 handler. When a packet is received on Endpoint 0, the USB drivers

check the request type for any USB level request packets. If the packet is not a standard USB request, it is passed up the stack via the usb_ep0_callback() which calls the us_ep0_hid callback().

The HID endpoint 0 handler processes the following HID specific requests.

```
/* Class specific requests. */
#define HIDRQ_GET_REPORT     0x1
#define HIDRQ_GET_IDLE       0x2
#define HIDRQ_GET_PROTOCOL   0x3
#define HIDRQ_SET_REPORT     0x9
#define HIDRQ_SET_IDLE       0xa
#define HIDRQ_SET_PROTOCOL   0xb
```

Data from host, interrupt driven, using Endpoint 0 (the control endpoint):



Data to the HOST is inserted directly into USB buffer descriptors for endpoint 1:

Synchronization with the USB bus is handled entirely by the USB module.

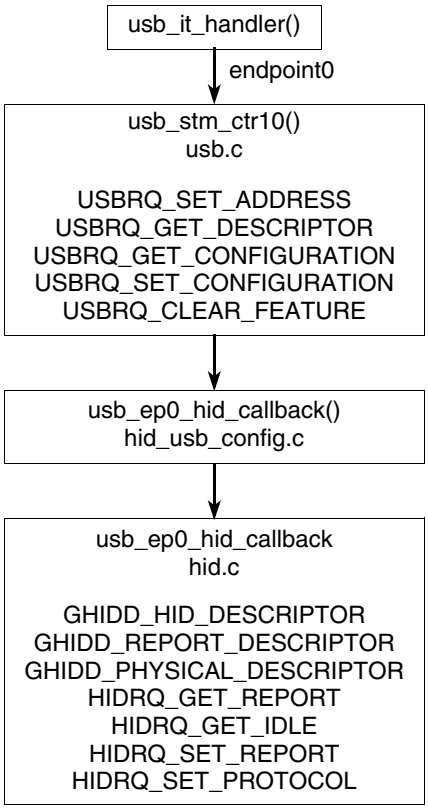


Figure 28. Request Propogation Through the Stack

The HID stack supports three type of reports: input, output, and features. The demo applications limit the number of reports to two (MAX_NO_OF_REPORTS in hid.c). The maximum number of bytes that can be sent through a report is limited to eight as defined by the macro MAX_REPORT_LENGTH in the file hid.c, and the report descriptors defined in the file hid_usb_config.c.

The report descriptors are in the file hid_usb_config.c. The format for report descriptors is defined in the HID specification. Report descriptors consist of a tag followed by parameters. There are two groups of tags, short and long. Short tags are limited to 1–5 bytes total length. Long tags can contain as many as 262 total bytes.

The total length of the report descriptor depends on the number and type of tags used in the descriptor.

To increase the maximum amount of data allowed in a report, the MAX_REPORT_LENGTH macro must be increased to increase the size of the report buffers, and the report descriptor must be modified so that the host is informed of the size of the report during enumeration.

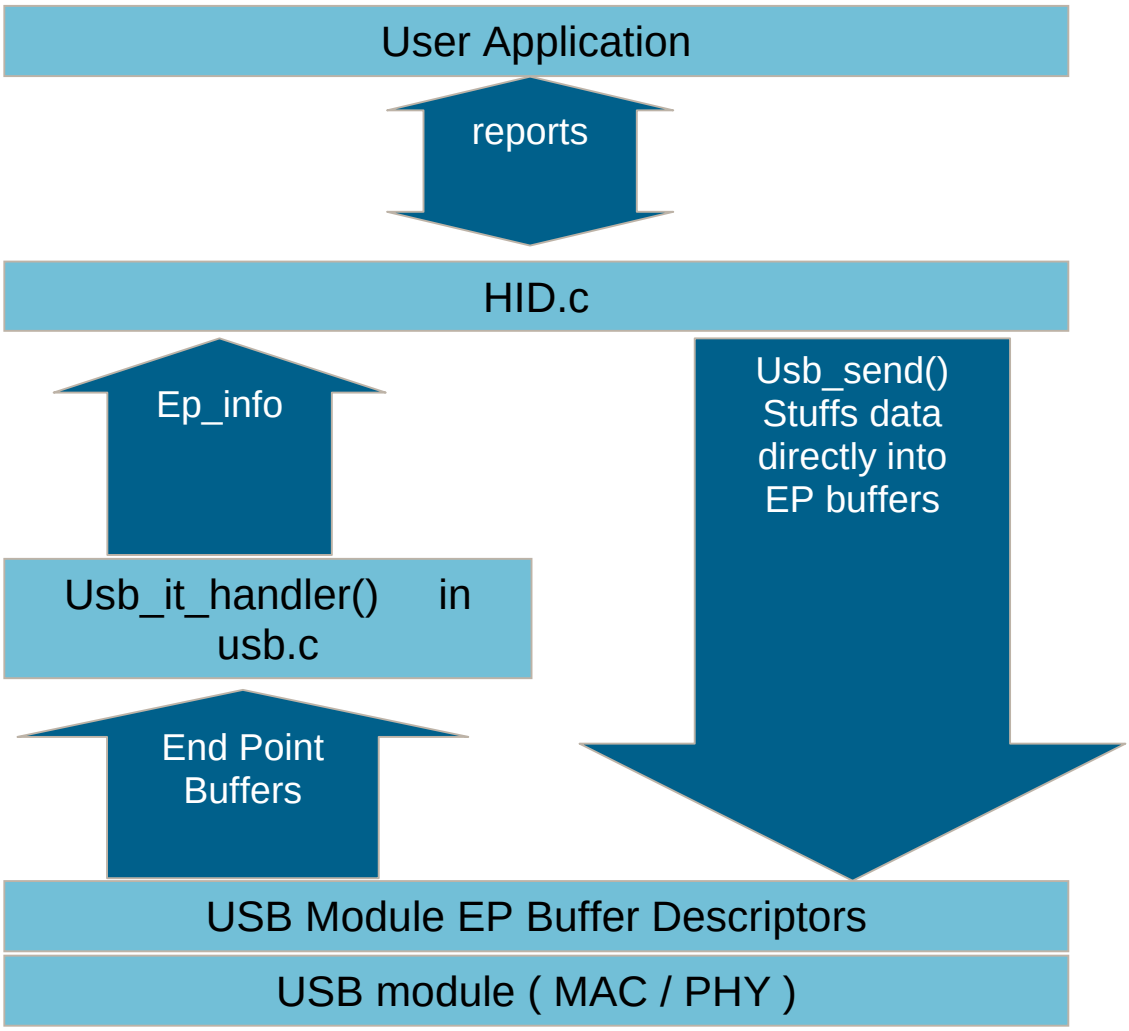


Figure 29. USB HID Stack Block Diagram

9.2 HID Device API

```
void HID_init(hcc_u16 default_idle_time, hcc_u8 ifc_number)
      default_idle_time is the idle time in ms reported back to the host via a
      HIDRQ_GET_IDLE request.
      Causes the endpoint to NAK any polls on an interrupt in
      endpoint while its current report remains unchanged. In the absence of a
      change, polling will continue to be NAKed for a given time-based
      duration.
      For keyboards, the spec recommends a idle time of 500ms, for mice the
      spec recommends a idle time of 0.
      The ifc_number specifies the interface number to start the HID protocol
      on.

void hid_process(void)
      This function walks through the reports and sends any reports marked as
      type rpt_in that are pending.  This is the transmit portion of the HID protocol.  Reports
      are transmitted on endpoint 1.

      for(x=0; x<sizeof(reports)/sizeof(reports[0]); x++)
      {
          if( reports[x].used
             && reports[x].pending
             && reports[x].type==rpt_in)
          {
                usb_send(1, (void *)0
                          , reports[x].buffer
                          , reports[x].size
                          , reports[x].size);
                while(usb_ep_is_busy(1));    // Wait for TX done
                reports[x].pending=0;
                break;
          }

      }
hcc_u8 hid_add_report(hid_report_type type, hcc_u8 id, hcc_u8 size)
      Initializes a unused report in the reports array.
      type is either: rpt_in, rpt_out, or rpt_feature as defined in hid.h
      id attaches this report to a REPORT ID created in a report descriptor.
      size is the number of byte in the report ( should match report descriptor ).
      Returns a report number to be used with write, read, and pending function
calls.

void hid_write_report(hcc_u8 r, hcc_u8 *data)
      Copy report.size ( set in add report ) bytes from *data into report r.
      The report is marked pending,  so it will be transmitted on the next
      call to hid_process().

void hid_read_report(hcc_u8 r, hcc_u8 *data)
      Copy report.size ( set in add report ) bytes from report r to *data.
      The report is marked not pending.

hcc_u8 hid_report_pending(hcc_u8 r)
      Returns the pending status of report r.
```

9.3 Sample HID Main Application Loop

```
void hid_generic(void)
{
      hcc_u8 out_report;
      hcc_u8 in_report;

      HID_init(500, 0);                                    // 500ms delay between report polls

      out_report=hid_add_report(rpt_out, 0, 1);            // reports from host,1 byte
      in_report=hid_add_report(rpt_in, 0, 1);              // report to host, 1 byte

      while(!device_stp)
      {
           hid_process();                                  // Send any pending rpt_in reports

           if (!hid_report_pending(in_report))             // Has the last report completed TX?
           {
                  hcc_u8 tmp=0;
                  hid_write_report(in_report, &tmp);     // Copy tmp to in_report
           }

           if (hid_report_pending(out_report))             // Has host sent a report?
           {
                  hcc_u8 rx_data;
                  hid_read_report(out_report, &rx_data);  // copy out_report to
                                                          // rx_data
           }
      }
}
```

10 PC-Side USB Host Software

To connect your embedded device to a PC host requires a driver on the PC side. The configuration descriptor is used by the device to inform the PC of which driver to load. Both Linux and Windows provide baseline drivers for the common USB classes.

10.1 Windows USB Drivers

The Windows Driver Kit (WDK) includes documentation for developing USB device drivers in a Windows environment. The Driver Development Kit (DDK) includes the libraries and examples of USB drivers. The DDK is included in the WDK. The Windows DDK requires Microsoft Visual Studio®. The DDKs include two USB driver examples, the BulkUSB and IsoUSB, plus a USB filter driver and a USBView utility. Early Win98/ME and Win2K examples are plagued by bugs, so it is recommended you use the WinXP DDK as a foundation for your new drivers.

Windows USB drivers are kernel-mode drivers. A kernel-mode driver runs in protected kernel space. The application cannot access protected kernel space directly. It uses a mechanism referred to as a I/O request packet, or IRP. A IRP is a data structure used to communicate between a driver in kernel space, other drivers in kernel space, or an application in user space.

The USB driver system contains many sub-drivers. These sub-drivers use IRPs to communicate between them. The application communicates with the USB stack using URB’s. USB client drivers set up USB

request blocks (URBs) to send requests to the host controller driver. The URB structure defines a format for all possible commands that can be sent to a USB device.

10.2 Windows XP® USB Driver Stack

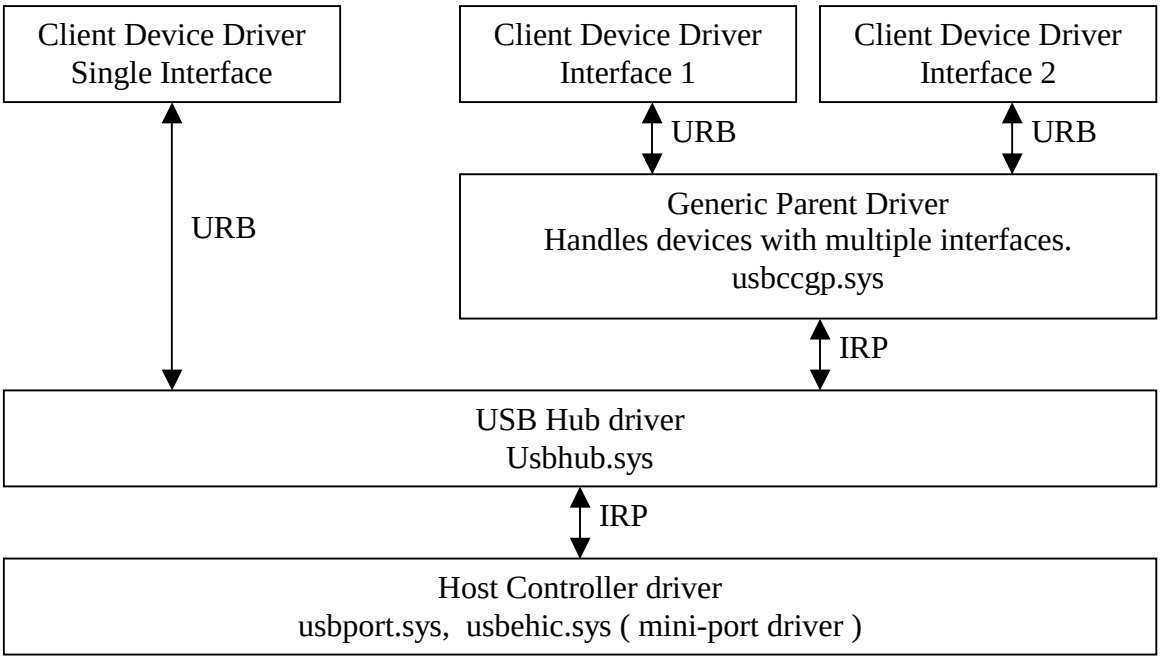


Figure 30. Windows XP USB Driver Stack

The host controller driver consists of the port driver, usbport.sys, and one or more of three miniport drivers that run concurrently. The port driver (usbport.sys) handles those aspects of the host controller driver's duties that are independent of the specific protocol.

The usbehic.sys mini-port driver is the hardware driver for enhanced host controller interface (EHCI) specific hardware.

The USB bus driver (usbhub.sys) is the driver for each USB hub. It is loaded if the PCI bus enumerator detects a USB hub. It exposes the USB driver interface.

Each particular device is supported by a USB client driver. Client device drivers for noncomposite devices are layered directly above the hub driver. For composite USB devices that expose multiple interfaces and do not have their own parent class driver, the system loads an extra driver called the USB common class generic parent driver, usbccgp.sys, between the hub driver and the client device drivers. The system assigns a separate PDO to each interface of a composite device.

Client device drivers for composite devices are loaded above the generic parent driver. A client driver for a composite device is no different from a client driver for a noncomposite devices, except for where it is loaded in the driver stack. Client drivers for composite devices sit above the generic parent driver.

Additional information about the USB driver stack can be found at www.msdn2.microsoft.com

10.3 Windows XP Supported USB Classes

Table 24. Windows XP Supported USB Classes

Class	Driver	Description
Hub Device	usbhub.sys	Used for managing USB hubs
Human Interface Device (HID)	hidclass.sys	Used to support USB HID standard devices.
Audio	sysaudio.sys	Audio class support
Mass Storage	usbstor.sys	For USB mass storage support
Printer	usbprint.sys	For USB printer support
Communication Device	usb8023.sys	Provides CDC class support
Imaging	usbscan.sys	manages USB digital cameras and scanners

10.4 Windows XP USB HID Support

The Windows DDK provides support for host-side communication with USB HID devices. The DDK includes documentation for the HID functions, and a overview on how to use them. HID application programming is done using these libraries and header files included in the DDK:

hid.lib

hidclass.lib

hidparse.lib

hidpi.h

hidsdi.h

hidusage.h

Using the hid libraries and example applications included in the SDK simplifies enumerating and communicating with HID devices.

10.5 Using the HID Library

1. /* Get the HID Globally Unique ID from the OS. */

```
GUID HidGuid;
HidD_GetHidGuid(&HidGuid);
```

2. /* Get an array of structures containing information about all attached and enumerated HIDs. */

```
HDEVINFO HidDevInfo;
HidDevInfo = SetupDiGetClassDevs(&HidGuid,
                                 NULL,
                                 NULL,
                                 DIGCF_PRESENT|
                                 DIGCF_INTERFACEDEVICE);
```

3. /* Get information about the HID device with the 'Index' array entry. */

```
Result = SetupDiEnumDeviceInterfaces(HidDevInfo,
                                     0, &HidGuid,
                                     Index,
                                     &devInfoData);
```

4. /* Get the size of the DEVICE_INTERFACE_DETAIL_DATA structure. The first call returns an error condition, but you will get the size of the structure. */

```
Result = SetupDiGetDeviceInterfaceDetail(HidDevInfo,
                                         &devInfoData,
                                         NULL,
                                         0,
                                         &DataSize,
                                         NULL);
```

5. /* Open a file handle to the device. Make sure the attributes specify overlapped transactions or the in transaction may block the input thread. */

```
hid_dev = CreateFile(    detailData->DevicePath,
                         GENERIC_READ | GENERIC_WRITE, /* read / write access*/
                         0,                            /* exclusive access */
                         NULL,                         /* No security */
                         OPEN_EXISTING,
                         FILE_FLAG_OVERLAPPED,         /* overlapped I/O */
                         NULL);
```

6. /* Get the device VID and PID to see if it is the device you want. */

```
if(hid_dev != INVALID_HANDLE_VALUE)
{
       HIDD_ATTRIBUTES HIDAttrib;
       HIDAttrib.Size = sizeof(HIDAttrib);
       HidD_GetAttributes( hid_dev, &HIDAttrib);
```

To write or read from the device, standard file I/O functions are used: writefile(), readfile(), and DeviceIoControl().

11 The Communication Device Class

The communication device class (CDC) is one class in a group of classes defined by the definition for communication devices. The other two classes are the communication interface class and the data interface class. The communication device class is a device- level definition and is used by the host to properly identify a communication device that may present several different types of interfaces. The communication interface class defines a general-purpose mechanism that can be used to enable all types of communication services on the USB. The data interface class defines a genera-purpose mechanism to enable bulk or isochronous transfer on the USB when the data does not meet the requirements for any other class.

Section 3.6.2.1, “Abstract Control Model Serial Emulation,” of the usbcdc version 1.1 specification defines serial emulation. A communication class interface is used for setting up the serial parameters (baud rate, data bits, flow control, etc.). The data class interface is used for transferring data.

11.1 CDC Serial to USB Configuration Descriptor

```
hcc_u8 usb_config_descriptor[] = {
      USB_FILL_CFG_DESC(9+9+5+4+5+5+7+9+7+7,   // descriptor size in bytes
                        2,                     // 2 interfaces
                        1,                     // configuration ID = 1
                        4,                     // 4 configuration strings
      CFGD_ATTR_SELF_PWR,                      // Device is self powered
                        0),                    // 0*2ma = 0 ma
      USB_FILL_IFC_DESC(0,                    // interface ID = 0
                        0,                     // no alternate setting
                        1,                     // This interface has 1 endpoint
                        2,                     // Class 2 = Communication Interface Class(28)
                        2,                     // SubClass 2 = Abstract Control Model(28)
                        1,                     // Protocol 1 = AT command set ??? ( page 28 )
                        5),                    // Description string 5
                        /* Comm interface/ abstract contorl model/ 00 no class protocol*/
      USB_FILL_HDR_FUNCT_DESCR(0x0110),        // Spec version 1.1 ( page 34 )
      USB_FILL_CALL_MGM_FUNCT_DESCR(0, 1),     // Capability = 00, Data Class Interface=1
      USB_FILL_ACM_FUNCT_DESCR(0),             // No ACM requests supported ( page 35 )
      USB_FILL_UNION_FUNCT_DESCR(0, 1),1,      // Master Interface =0, Slave = 1( page 40 )
      USB_FILL_EP_DESC(0x1,                    // address = 0
                EPD_DIR_TX,                    // IN endpoint ( device to host )
               EPD_ATTR_INT,                   // Interrupt Endpoint
            EP1_PACKET_SIZE,                   // 32 bytes
                        0x2                    // 2ms poll interval
      USB_FILL_IFC_DESC( 1,                    // Interface ID = 1
                         0,                    // No alternate setting
                         2,                    // This interface has 2 endpoints
                         10,                   // Data Interface Class
                         0,                    // Subclass 0
                        0x0,                   // Protocol 0
                         6),                   // Description string 6
                          /* Data interface/0/ no class specific protocol. */
      USB_FILL_EP_DESC(0x2,                    // Address  = 2
                EPD_DIR_TX,                    // IN endpoint ( device to host )
              EPD_ATTR_BULK,                   // Bulk transfer
            EP2_PACKET_SIZE,                   // 32 bytes
                        0x0),                  // Ignored for Bulk endpoints
      USB_FILL_EP_DESC(0x3,                    // Address = 3
                EPD_DIR_RX,                    // OUT endpoint ( host to device )
              EPD_ATTR_BULK,                   // Bulk transfer
            EP3_PACKET_SIZE,                   // 32 bytes
                        0x0),                  // Ignored for Bulk endpoints
};
```

The demo CDC to serial firmware uses three endpoints;, two configured for bulk transfer (one in, one out). These endpoints are used to exchange data. The third endpoint (address 1) is a interrupt transfer endpoint used to transfer configuration data.

11.2 CDC Specific Requests

Requests are methods or functions that the host calls to initiate a action on the device. The CDC requests are used to adjust the serial parameters. The requests are sent down the default pipe, Endpoint 0.

Table 25. CDC Resquest Descriptions

Request	Descriptor
GET_LINE_CODING	Requests current DTE rate, stop-bits, parity, and number-of-character bits.
SET_LINE_CODING	Configures DTE rate, stop-bits, parity, and number-of-character bits.

11.3 CDC Serial to USB Demonstration Firmware

The CDC project uses the usb.c driver. The configuration data described above is located in the file cdc_usb_config.c. The file usb_cdc.c contains the CDC specific methods and USB driver callbacks. The Endpoint 0 callback function is used to capture the CDC specific requests.

The following code is a snippet of the Endpoint 0 callback function from the CDC demo. The function handles the two CDC specific requests. When the host sends a SET_LINE_CODING request, the firmware setups a received ep_info structure by calling the usb_receive() function. The receive expects seven bytes from the host, and directs the driver to put the bytes into the line_coding structure. The receive also sets the function got_line_coding() to be called after the data is received by the device. This function sets the new_line_coding flag to indicate to the rest of the firmware that new serial parameter data is available.

The line_coding[] array is a structure defined on page 58 of the CDC specification. The function cdc_get_line_coding(line_coding_t *l) translates.

Byte	0 1 2 3	4	5	6
	BAUD	STOP BITS	PARITY	# OF DATA BITS

```
callback_state_t usb_ep0_callback(void)
{
      hcc_u8 *pdata=usb_get_rx_pptr(0);

      /* A request to the command interface. */
      if (STP_INDEX(pdata) == CMD_IFC_INDEX)
      {
          switch(STP_REQU_TYPE(pdata))
          {
               /* Class specific in request. */
               case ((1<<7) | (1<<5) | 1):
                    /* Host wants to get a descriptor */
                    switch (STP_REQUEST(pdata))
                    {
                         case CDCRQ_GET_LINE_CODING:
                              usb_send( 0,
                                  (void *) 0,
                                  (void *)&line_coding,
                                           7,
                                  STP_LENGTH(pdata));
                              r=clbst_in;
                              break;
                         default:
```

```
                        break;
                }
                break;

            /* Class specific out request. */
            case ((0<<7) | (1<<5) | 1):
                switch (STP_REQUEST(pdata))
                {
                    case CDCRQ_SET_LINE_CODING:
                        usb_receive(0,
                                    got_line_coding,
                                    (void *)&line_coding,
                                    7,
                                    7);
                        r=clbst_out;
                        break;
                    default:
                        break;
                }
                break;
    }
}
```

11.4 CDC Serial to USB API

void cdc_init(void)

Initialize the CDC buffers.

void cdc_get_line_coding(line_coding_t *l)

Translate the CDC line coding structure into a line_coding_t structure.

```
typedef struct {
                    hcc_u32 bps;
                    hcc_u8 ndata;
                    hcc_u8 nstp;
                    hcc_u8 parity;
                } line_coding_t;
```

int cdc_line_coding_changed(void)

The application should call this function routinely.

Returns the new_line_coding flag, then clears the flag.

Returns 1 to indicate the host has sent a new line_coding structure.

int cdc_putch(hcc_u8 c)

Inserts the byte c into a tx buffer. The character will be sent the next time the host polls the device. This function calls usb_send().

Returns 1 if the character is successfully buffered.

Returns 0 if buffer was not available.

int cdc_input_ready(void)

The application should call this function routinely.

Uses the usb_receive() function to setup a ep_info structure to receive data on the RX_EP_NO endpoint.

Returns 1 if data has been received from the host, and is ready to be read by the application using a cdc_getch().

Returns 0 if no data is ready.

char cdc_getch(void)

Returns the next character from the rx_buffer.

This function should only be called if cdc_input_ready() returns 1.

callback_state_t usb_ep0_callback(void)

USB driver controls Endpoint 0 callback function. This function intercepts the CDC specific requests: CDCRQ_GET_LINE_CODING, and CDCRQ_SET_LINE_CODING.

callback_state_t got_line_coding(void)

This function is called by the control Endpoint 0 interrupt after new data is received by the host. Data from the host on endpoint 0 is configuration data for the serial port.

11.5 Example CDC Application

```
int main()
{
      hw_init();
      /* USB irq is level 2, priority = 2. */
      usb_init((2<<3) | 2, 0);
      /* UART irq is level 3, priority = 3. (UART needs higherpriority. */
      uart_init(9600, 1, 'n', 8);
      cdc_init();

      /* Main application Loop */
      while(1)
      {
          // Check for serial config change from host
          if (cdc_line_coding_changed())
          {
                 // Configuration change
                 line_coding_t l;
                 hcc_u8 parity[]="noe";
                 cdc_get_line_coding(&l);
                 uart_init(l.bps, l.nstp, parity[l.parity], l.ndata);
          }

          // Has the host sent data
          if (cdc_input_ready())
          {
                 // Get character from CDC buffer and copy to UART buffer
                 char c=cdc_getch();
                 uart_putch((hcc_u8)c);
          }

          // Is there any UART data
```

```
            if (uart_input_ready())
            {
                  // Read data from UART, send to USB
                  hcc_u8 c=uart_getch();
                  cdc_putch(c);
            }
      }
      return 0;
}
```

12 USB On-The-Go

USB On-The-Go (OTG) is defined by the On-The-Go supplement to the USB2.0 Specification. The purpose of OTG is to allow a peripheral to be a host or a device. A good application for OTG is a camera. After the pictures are taken, the camera is plugged into a PC to download the pictures. In this case, the camera is a device, but he user can also plug the camera directly into a printer, to print the pictures directly. In the second case, the camera acts as a host.

A OTG device is either a host or a device, its mode is determined when it is connected to another USB product. The host negotiation protocol (HNP) is used to negotiate who is host. Another protocol, the session request protocol, determines who will power the bus.

12.1 Terminology

A-Device — A device with a Standard-A or Micro-A plug inserted into its receptacle. The A-device supplies power to VBUS and is host at the start of a session. If the A-device is OTG (equipped with a Micro-AB receptacle), it may relinquish the role of host to an OTG B-device under certain conditions.

B-Device — A device with a Standard-B, Micro-B or Mini-B plug inserted into its receptacle, or a captive cable ending in a Standard-A plug. The B-device is a peripheral at the start of a session. If the B-device is OTG (equipped with a Micro-AB receptacle), it may be granted the role of host from an OTG A-device

12.2 Session Request Protocol (SRP)

The session request protocol (SRP) allows a B-device to request the A-device to turn on VBUS and start a session. OTG-B (device) asks OTG-A (host) for a USB OTG session by signaling in one of two ways:

1. Pulsing an analog data line (D+ or D–)
2. Pulsing VBUS through a relatively high impedance (> 281 ohm)

The OTG-B (device) pulses data line first then pulses VBUS.

Per the OTG specification: “Any A-device, including a PC or laptop, is allowed to respond to SRP. Any B-device, including a standard USB peripheral, is allowed to initiate SRP. An On-The-Go device is required to be able to initiate and respond to SRP.”

12.3 Host Negotiation Protocol

Host negotiation protocol (HNP) allows the host function to be transferred between two directly connected OTG devices. HNP may only be implemented through the Micro-AB receptacle on a device.

1. OTG-A (host) enables OTG-B (device) to become the host by sending SetFeature (b_hnp_enable) command to OTG-B (DEVICE).
2. OTG-A (host) suspends bus signaling so that OTG-B (device) can become host.
3. OTG-B (device) detects suspend condition and turns off pullup resistor.
4. Because HNP is enabled, OTG-A (host) interprets this disconnect as a request by the OTG-B (device) to become host.OTG-A (HOST) turns on its pull-up resistor and becomes peripheral/device
5. To return control back to OTG-A (host), OTG-B (device) stops using bus and becomes peripheral/device.
6. OTG-A (host) sees lack of activity, disconnects, and becomes the host.

If the OTG-B (device) does not stall the SetFeature(b_hnp_enable) command, the OTG-A (host) must give the OTG-B (device) an opportunity to become the host before the OTG-A (host) may turn off VBUS.

13 Resource Usage

The USB stack uses a minimum amount of RAM and flash. The numbers below represent values pulled from the MAP files of the actual demo projects listed. These are baseline demos as distributed in the zip file from the Freescale website. The stack RAM usage is as specified in the original distributed firmware.

The hid-demo-flash project supports a keyboard device, a mouse device, and a generic device with the numbers listed. The primary impact of supporting all three devices is in flash size. Supporting only one device, removing support for the other two, would have a very small (less then 16 bytes) impact on RAM requirements. Three different configuration descriptors are stored in flash. Only one is required. Each group of descriptors is approximately 300 bytes, so if you are building a mouse device only, or a keyboard device only, or a generic device only, subtract 600 bytes from the flash number shown below for the hid-demo-flash project.

Table 26. CMX USB Stack Memory Usage

	HID device	CDC device	OTG	HID host	Mass storage Host
	hid-demo-flash	cdc-demo-flash	otg-app	host-hid-demo	mass-storage-demo
flash	22960	18832	54128	23904	35728
RAM (total)	7680	7168	11264	7168	7680
stack	5120	5120	7168	5120	5120
bss	1621	1224	3338	1404	1760
bdt+align	939	824	758	644	800

Table 27 was generated after testing the firmware with smaller stack sizes. The new stack sizes were tested by putting a marker at the end of the stack and running the demo through its paces. Demo operation was verified, and after the test the marker was checked (confirming that the stack did not overrun past the end).

Table 27. CMX USB Stack Memory Usage (Reduced Call Stack)

	HID device	CDC device	OTG	HID host	Mass Storage Host
	hid-demo-flash	**cdc-demo-flash**	**otg-app**	**host-hid-demo**	**mass-storage-demo**
flash	22960	18832	54128	23904	35728
Ram (total)	**4608**	**4096**	11264	**4096**	**4608**
stack	**2048**	**2048**	7168	**2048**	**2048**
bss	1621	1224	3338	1404	1760
bdt+align	939	824	758	644	800

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3492
Rev. 0
08/2007

freescale™
semiconductor