Freescale Semiconductor
Application Note

Document Number: AN3153
Rev. 0, 08/2006

Using the Full-Speed USB Module on MC68HC908JW32

Including USB Basics, Driver Description, Software Examples, CDC — Virtual Serial COM, and HID Class

by: Radomir Kozub
Freescale Roznov pod Radhostem Czech Republic

The universal serial bus (USB) has become the most widespread periphery among PCs. Because embedded devices need to communicate with personal computers, USB is the ideal solution.

This document provides instructions for creating your first USB device. MC68HC908JW32 has an advanced USB 2.0 full-speed module that enables communication with a PC through USB with minimal software or hardware efforts. This document contains a description of MC68HC908JW32, suggestions for its use, and a discussion of USB basics. The embedded USB module speeds software design, and a description of the driver and examples for using it are included. In addition, this document discusses the communication device class (CDC) — virtual serial COM — that eases migration from UART to USB and human interface device (HID) class and contains an example of the implementation of a mouse.

Contents

1 MC68HC908JW32 USB Module Description

1.1 USB Module Features

The USB module complies with the *Universal Serial Bus Rev. 2.0 Specification* full-speed functions:

- 12 Mbps data rate
- On-chip 3.3 V regulator
- Endpoint 0 with an 8-byte transmit buffer and an 8-byte receive buffer
- Four data endpoints supported by a shared 64B in/out buffer
- Programmable endpoint type for four independent endpoints — interrupt or bulk
- USB device controller with protocol control supporting a single configuration, two interfaces with no alternative settings for either interface
- USB data control logic/packet identification and decoding/generation, CRC generation and checking, non-return-to-zero inserted (NRZI) encoding/decoding, bit-stuffing, sync detection, end-of-packet detection
- USB reset options: internal MCU reset generation and CPU interrupt request generation

1.2 USB Driver Files

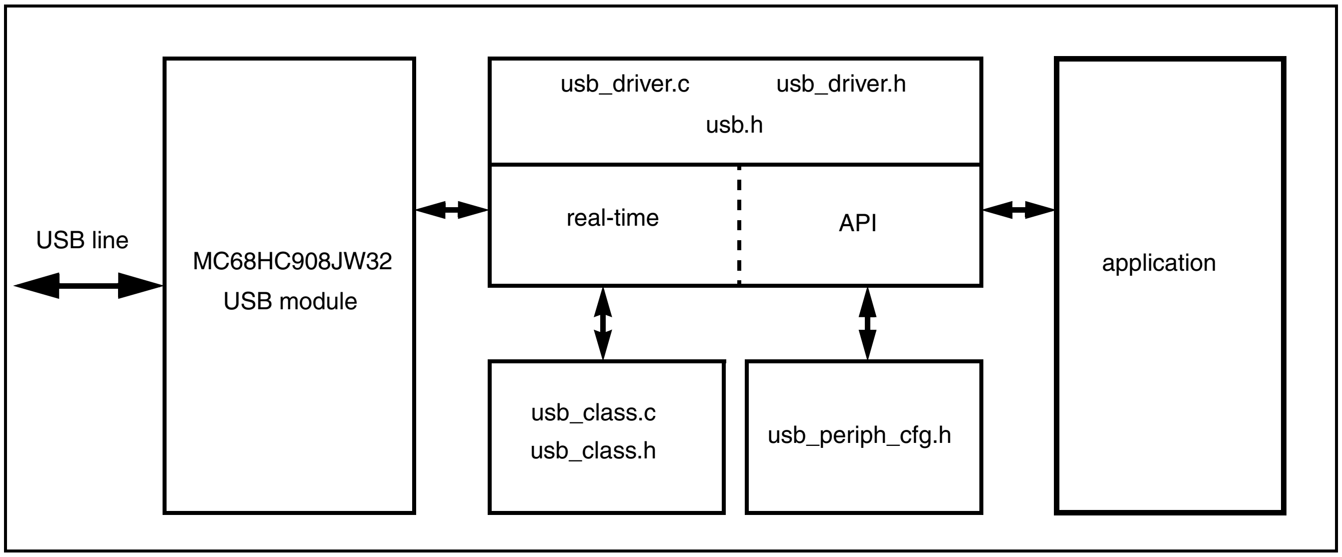

Figure 1. USB Driver Files Structure

The USB driver resides in usb_driver.c, usb_driver.h, usb.h. These files should not be changed.

The static settings by macro constants of the USB module for an application are defined in the usb_periph_cfg.h file. Most of these initializations, stated in usb_periph_cfg.h, are used in the driver usb_driver.c file for initializing the USB module (endpoint parameters, buffers, interrupts, etc.), and also for code tailoring during compilation time (conditional code). Section 1, “MC68HC908JW32 USB Module Description,” and Section 2, “USB Interrupts,” describe all the necessary static settings.

The usb.h file contains data structures defined in *Universal Serial Bus Rev. 2.0 Specification* that are used mostly in the driver, such as the device, configuration, endpoint and interface descriptor format, setup packet data structure, etc.

Driver code is in the usb_driver.c file. By using macro constants from usb_periph_cfg.h, the source code is tailored during the compile time and no changes are needed.

In the HID example, you can find the application dependent files *usb_hid.c* and *usb_hid.h* (USB class files) where all the needed descriptors lie. User code must be placed there to manage vendor and device specific requests.

1.3 Electrical Basics

USB is a host-controlled serial bus that uses a star topology. All communication is initiated by the host — all devices are polled. Only one host exists in any USB system. USB devices are hubs or functions. Hubs provide additional attachment points to the USB host. Functions provide capabilities to the system, such as an ISDN connection, a digital joystick, or speakers.

The USB transfers signals and power over a four-wire cable with twisted pairs as specified in the *Universal Serial Bus Rev. 2.0 Specification*, Chapter 7.1.1.1. Data is signalled differentially over D+ and D–. A NRZI encoding scheme is used with a sync field to synchronize the host and receiver clocks. The other two wires, 5 V power lines Vbus and GND, are assigned to supply devices.

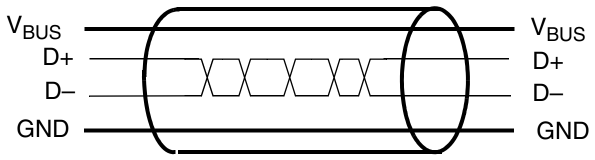

Figure 2. USB Lines — Data (D+, D–) and Power Supply (V_{Bus}, GND)

1.4 Power the Device by USB

One of the advantages of USB is bus-powered devices, which receive their power from the bus and require no external plug packs.

USB specification uses three types of USB functions:

- Low power bus-powered functions (max 100 mA)
- High power bus-powered functions (max 500 mA)
- Self-powered functions (max 100 mA from the bus, rest from an external source)

More details can be found in the *Universal Serial Bus Rev. 2.0 Specification*, Chapter 7.2.1.

Because MC68HC908JW32 has an operating voltage range of 3.5 to 5.5 V, it can be powered directly by USB without using the LDO regulator (see Figure 3). When you plug a USB device into the USB connector, the MCU supply voltage raises and it starts to work. Then you can initialize the MCU and its USB module. When the USB module is ready, enable a speed select pullup resistor (R_{PU}) to let the host know that the device is attached to the bus.

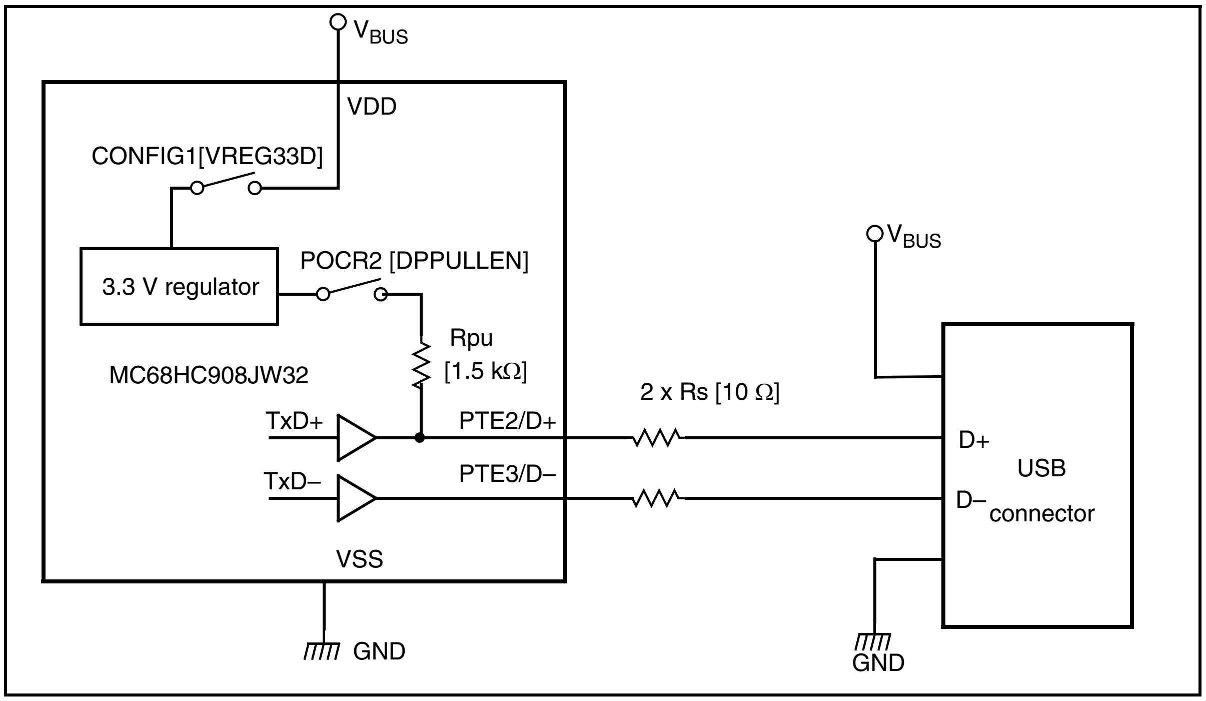

Figure 3. Connecting MC68HC908JW32 to USB

1.5 USB Speed Selection

Chapter four in *Universal Serial Bus Rev. 2.0 Specification* describes three data rates:

- The USB high-speed signaling bit rate is 480 Mbps
- The USB full-speed signaling bit rate is 12 Mbps
- A limited capability low-speed signaling mode is also defined at 1.5 Mbps

USB lines are terminated at the hub and function ends as shown in Figure 4. Full-speed, high-speed, and low-speed devices are differentiated by the position of the pullup resistor on the downstream end of the cable. Full-speed and high-speed devices are terminated with the pullup resistor Rpu (1.5 kΩ) on the D+ line to 3.3 V. Low-speed devices have a pullup resistor on the D– line. Both full-speed and high-speed devices have a pullup resistor on the D+ line and they are differentiated by a high-speed chirp during reset.

MC68HC908JW32 USB module supports only full-speed; therefore, this discussion concerns the full-speed mode only.

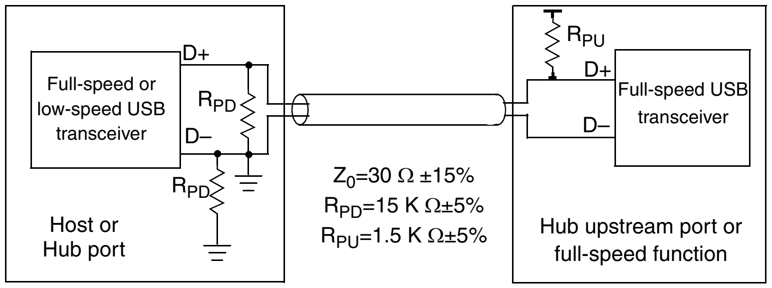
Full-speed or low-speed USB transceiver
D+
D–
R_{PD}
R_{PD}
Host or Hub port
Z_0=30 Ω ±15%
R_{PD}=15 K Ω±5%
R_{PU}=1.5 K Ω±5%
R_{PU}
D+
D–
Full-speed USB transceiver
Hub upstream port or full-speed function

Figure 4. Full-speed Device Cable and Resistors Connection

MC68HC908JW32 has an integrated voltage regulator that provides 3.3 V output. It can be connected through an integrated pullup resistor to the D+ line (see Figure 3).

The 3.3 V voltage regulator can switched on or off by the CONFIG1[VREG33D] bit. The integrated pullup resistor (1.5 kΩ) between the 3.3 V output and the D+ line can be enabled or disabled by the POCR[DPPULLEN] bit.

If no pullup resistor is connected to the D+ or D– lines, the USB assumes nothing is connected to the bus.

1.5.1 Programming

As discussed in Section 1.2, all static settings used in the driver are set up by macro constants in the usb_periph_cfg.h file.

The 3.3 V voltage regulator is enabled by default on the MC68HC908JW32. To disable it, comment out the macro constant below. The voltage regulator will be disabled within:

```
#define USB_3V3REGULATOR_DIS  0x02
```

The internal pullup resistor (D+ to 3.3 V) is disabled by default. To enable it, define the macro constant:

```
#define USB_PULLUP_ENA 0x01
```

in usb_periph_cfg.h

Both settings are employed within the USB_Init() function in usb_driver.c. After the MCU reset, all module setup is done in this function also. See Section 2.1 for more discussion.

Call the function USB_Deinit() to disable the D+ pullup resistor. This will signal the host that the USB function was deasserted.

1.6 Endpoint and Pipe

The MC68HC908JW32 USB module contains five endpoints: EP0, EP1, EP2, EP3, and EP4. Endpoints are a source or sink of data located at the end of a logical communication channel, known as a pipe, to the USB function. A host can send or receive data to or from an endpoint. Each endpoint can be a source or a sink of data. Each endpoint has its own buffer of defined length.

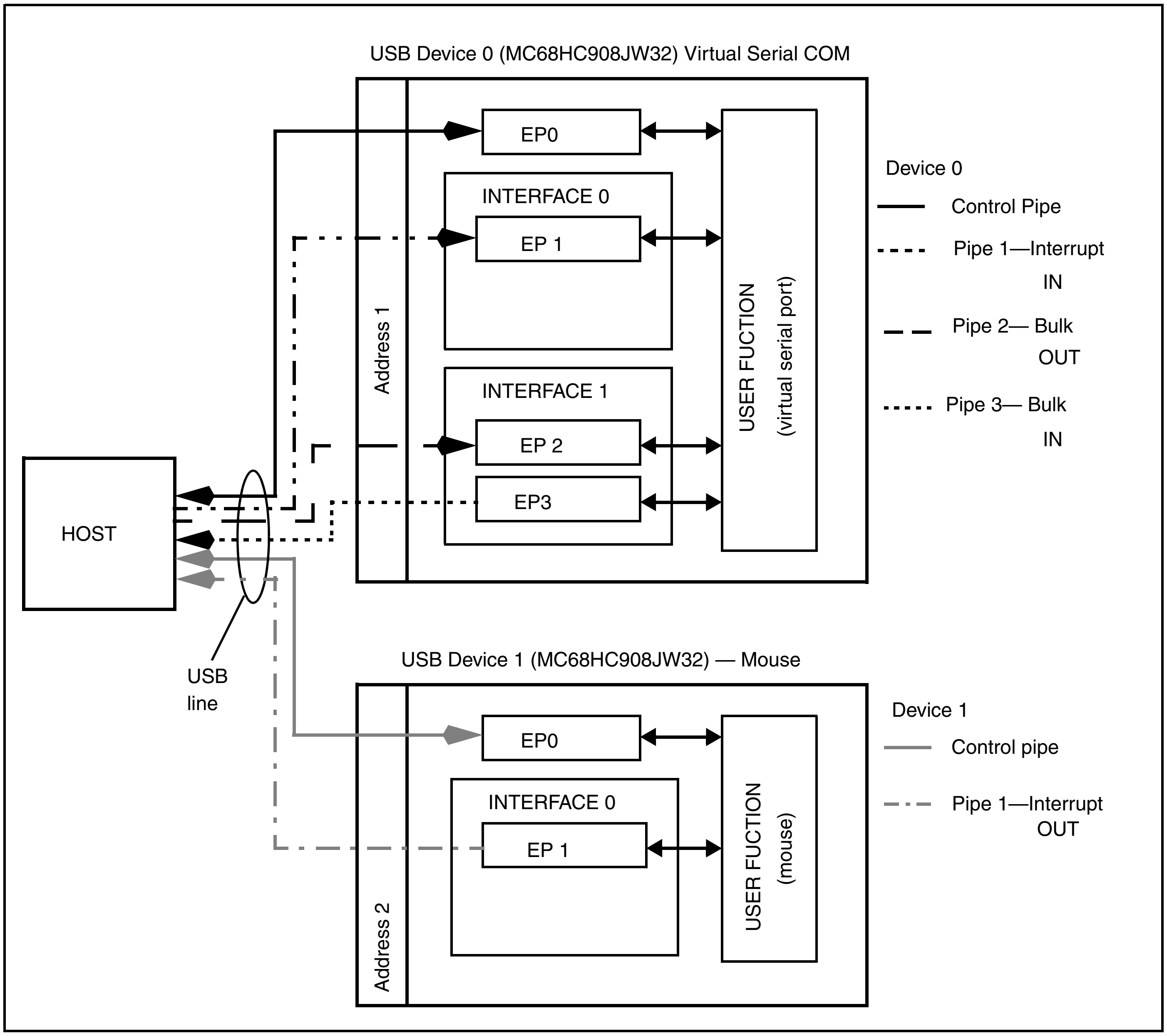

Figure 5. USB Logical Structure — Endpoints and Pipes

Each endpoint used in the USB device is logically connected to the host by a pipe. The pipe is a logical communication channel between the USB host and device. It has a set of parameters associated with it, such as dataflow direction (IN, OUT), transfer type (bulk, interrupt) or maximal packet size.

As shown in Figure 5, each USB device contains a control pipe. On the MC68HC908JW32, the default control pipe is made up of a bidirectional endpoint zero, EP0. It uses control transfer, both IN/OUT directions, and has a dedicated 8-byte bidirectional buffer and packet size at address $0040–$0047. EP0 is the only bidirectional endpoint on the MC68HC908JW32. EP0 does not need to be set because the setting is always the same and creates the control pipe. EP0 is set in the USB_Enable() function. Also, the EP0 data transfer interrupt is enabled by default.

The endpoints and their pipes have a set of parameters that must be defined before communicating over the USB module. Most of them are set in USB endpoint control and status register, UEPxCSR. Parameters cannot be changed after the USB module is enabled as the bits have no effect.

1.6.1 Endpoint Direction

Endpoints EP1 to EP3 must have their data direction set as either IN or OUT, as opposed to bidirectional EP0. An IN endpoint has logical data flow from the USB device to the host, however an OUT endpoint receives data from the host. Endpoint direction is set by bit UEPxCSR[DIR] before the USB module is enabled and cannot be changed later.

1.6.1.1 Programming

Endpoints EP1 to EP3 direction macro constants must be initialized by one of the predefined values (EP_DIR_IN, EP_DIR_OUT) in the usb_periph_cfg.h file. These macro statements are employed in the USB_Enable() function. The endpoint should have the same direction in the endpoint descriptor:

```
#define EP1_DIR                  EP_DIR_IN
#define EP2_DIR                  EP_DIR_OUT
```

These definitions are also used in the compilation time to select the correct interrupt service routine in usb_driver.c if the interrupt service routine (ISR) is enabled.

1.6.2 Endpoint Mode

Endpoints EP1 to EP3 are data endpoints used to transfer data from the USB device to the USB host and vice versa. Using bits UEPxCSR[MODE], you can set the endpoints as bulk or interrupt (the endpoint will use a bulk or interrupt data transfer). Other transfer types such as control and isochronous, are not allowed. As the host polls the USB devices, the interrupt endpoint must wait until the host asks it. Interrupt transfer guarantees communication latency; however, bulk transfer uses spare unallocated bandwidth on the bus after all other transactions have been allocated. After the USB module is enabled, the transfer type cannot be changed.

Table 1. Endpoint Mode Selection

MODE[1:0]	Endpoint Type
00	Disable
10	Bulk
11	Interrupt

EP0 is dedicated to control transfer and a different setting is not possible.

1.6.2.1 Programming

Each endpoint mode macro constant (EP1_MODE), except EP0, must be initialized by one of the predefined values: EP_MODE_INT, EP_MODE_BULK or EP_MODE_DISABLE, in the usb_periph_cfg.h. Macro constants are employed in the USB_Enable() function to fill the UEPxCSR registers. These macro statements might also be used in the USB descriptors. The endpoint mode must be the same as that defined in the endpoint descriptor.

```
#define EP1_MODE                EP_MODE_INT      // Endpoint 1 will be initialized by EP1_MODE
#define EP2_MODE                EP_MODE_BULK
#define EP3_MODE                EP_MODE_DISABLE
#define EP4_MODE                EP_MODE_DISABLE
```

1.6.3 Buffer Size

EP1 to EP3 share one 64-byte IN/OUT buffer located at address $1000–$103F. Certain parts of the buffer space must be allocated to each used endpoint by the UEPxCSR[SIZE] bits.

Table 2. Endpoint Buffer Size Selection

SIZE[1:0]	Buffer Size
00	8 Bytes
01	16 Bytes
10	32 Bytes
11	64 Bytes

In registers UEP12BPR and UEP34BPR you must set the endpoint buffer base address within the buffer. See the MC68HC908JW32 USB module user's guide.

1.6.3.1 Programming

Each EP1 to EP3 buffer size macro constant (EP1_BUF_SIZE_BITMAP) should be initialized by one of the predefined values (EP_BUFFER_SIZE_0, EP_BUFFER_SIZE_8, EP_BUFFER_SIZE_16, EP_BUFFER_SIZE_32, EP_BUFFER_SIZE_64) in the usb_periph_cfg.h file as seen below:

```
#define EP1_BUF_SIZE_BITMAP    EP_BUFFER_SIZE_8 // endpoint 1 has an 8 byte buffer
#define EP2_BUF_SIZE_BITMAP    EP_BUFFER_SIZE_16
#define EP3_BUF_SIZE_BITMAP    EP_BUFFER_SIZE_16
#define EP4_BUF_SIZE_BITMAP    EP_BUFFER_SIZE_0 // disabled endpoint has zero length buffer
```

These macro constants initialize the UEPxCSR registers within the USB_Enable() function. Also, the endpoint buffer size constants (EPx_BUF_SIZE_BITMAP) are used directly to calculate endpoint buffer base addresses loaded into UEP12BPR and UEP34BPR registers within USB_Enable(). EP1 always has its buffer base address pointer set to $1000. EP2 has its buffer base address pointer set to the value $1000 + EP1_BUFFER_SIZE, and so on.

EPx_BUFFER_SIZE macro constants are calculated from EP1_BUF_SIZE_BITMAP by a preprocessor in the usb_periph_cfg.h file, and must be used in the USB driver functions as endpoint max packet length, and also in USB descriptors as packet length. The buffer size assigned to the endpoint is required to match the size defined in the endpoint descriptor.

1.7 STALL

Setting the STALL bit will cause an endpoint to send a STALL handshake when polled by either an IN or OUT token from the USB host. A STALL handshake is issued to the host if the endpoint is halted or some request is not supported (EP0 only). Refer to the MC68HC908JW32 Data Sheet for details on latching halt state on an endpoint. A STALL is not a static setting of the USB module, it is used in runtime.

Use the macro constant USB_WRSTALL_EPx(x) to set/clear the STALL bit.

2 USB Interrupts

The MC68HC908JW32 has two interrupt sources for the USB module. USB endpoint interrupt is called if a successful transaction is finished between host and function endpoint. It is primarily used during data transfer.

USB system interrupt is concerned more with the states of the USB module or line. It can be generated from different sources — reset, suspend, resume, by an SOF and setup packet or config change request. Each source of the USB status interrupt has its own interrupt enable and interrupt flag bits. You must define the macro constant described in the subsequent sections to let the driver enable the interrupt (within USB_Enable()), and add service code to the interrupt service routine (ISR). In the ISR the interrupt flag is cleared and the call back function is called. You must write your own call back functions.

2.1 USB System Interrupts

2.1.1 USB Line Reset State

The USB host issues the reset state on the USB line by pulling the D+ line to logic zero for at least 2.5 μs (10 ms required).

Reset state is generated online by the host to put the function into a default state during the enumeration process after power up. The reset state is detected by the USB module. The MC68HC908JW32 MCU has two options for managing it:

If bit CONFIG2[URSTD] is logic 0, the default state after MCU reset, the USB line reset state recognized by the USB module will generate an internal reset of the MCU.

If bit CONFIG2[URSTD] is logic 1, then the USB line reset generates an interrupt request to the CPU.

To disable the MCU internal reset and enable a CPU interrupt request, define a macro constant

```
#define USB_CHIP_RESET_DIS    0x01      in file usb_periph_cfg.h
```

If a USB line reset generates interrupt request, you must enable or disable the USB RESET interrupt by setting or clearing bit USIMR[USBRESETIE]. The USB RESET interrupt will be enabled within the USB_Enable() function and the service code will be added to the ISR if the macro constant

```
#define USB_USBRSTIE_ENA     1    is defined in the file usb_periph_cfg.h
```

In the ISR USB_SYS_ISR, the flag USBSR[USBRST] is cleared and the callback function USB_ResertCB() is called.

2.1.2 Suspend and Resume

If USB lines are idle for at least 3 ms, the USB module detects a suspend state**.** When the function detects a suspend state it has 7 ms to decrease power consumption, for example, switch off all peripherals, stop the PLL to decrease MCU bus clock, or go into stop mode. Stop the USB module clock by clearing bit USIMR[USBCLKEN] before putting the MCU into a stop. The maximum suspend current is proportional to the allowed function load that is stated in the descriptor. For a 100 mA load device, the maximum suspend current is 500 µA. By defining the macro constant

```
#define USB_SUSPNDIE_ENA     1    in file usb_periph_cfg.h
```

the SUSPNDIE interrupt is enabled within USB_Enable(). Callback function USB_SuspendCB() is called and the SUSPND flag is cleared during an interrupt service routine. You must write the USB_SuspendCB() function. The suspend state is used when the host goes to sleep and wants to prevent the battery from discharging through the USB module.

The device will resume from the suspend state when it receives any non-idle signalling during suspend mode. If the macro constant

```
#define USB_RESUMEFIE_ENA    1
```

is defined in file usb_periph_cfg.h then bit USBSR[RESUMEF] is set and the RESUME interrupt is enabled within the USB_Enable() function. USB_ResumeCB() function is called and the resume flag RESUMEF is cleared during an interrupt service routine. You must write the USB_ResumeCB() function.

The USB function can force a resume state onto the USB bus data lines to initiate a remote wakeup by setting bit USBCR[RESUME]. The resume state is issued only if the function is in suspend mode and the remote wakeup feature is enabled by the SET_FEATURE command. Use the macro statement USB_RESUME() to force the resume state on the USB line.

2.1.3 Config Change

CONFIG CHANGE interrupt is enabled by setting bit USIMR[CONFIG_CHGIE]. To do that, define

```
#define USB_CONFIGCHGIE_ENA  1
```

in file usb_periph_cfg.h and the interrupt will be enabled within the USB_Enable() function.

During enumeration, CONFIG_CHANGE is the last step to have a function ready from the USB point of view. In the configuration descriptor there is a configuration value to which the device is set by a SET_CONFIG request. When a SET_CONFIG request is received, a CONFIG CHANGE interrupt is generated. Callback function USB_ConfigChngCB(uchar cfgNum) is called and USBSR[CONFIG_CHG] is cleared.

When the USB device is configured, the function can start communicating over the USB.

2.1.4 SOF

Start of frame (SOF) packet is issued on full-speed USB every 1 ms. The MC68HC908JW32 USB module can recognize it and generate a START OF FRAME interrupt. Define the macro constant

```
#define USB_SOFIE_ENA        1
```

in the usb_periph_cfg.h file to enable a SOF interrupt in the USB_Enable() function.

Although the SOF packet contains an 11-bit length frame number data (see Section 4.1), the USB module does not provide this data. In the SOF ISR, the USB_SofCB() function is called and the SOF flag cleared.

2.1.5 Setup

When a setup token packet is received and the standard request is not managed by the USB module request processor, then the setup flag USBRST[SETUP] is set and an interrupt is generated if bit USIMR[SETUPIE] is set. The SETUP interrupt will be enabled within the USB_Enable() function if the macro constant

```
#define USB_SETUPIE_ENA      1
```

is defined in the usb_periph_cfg.h file, otherwise it is disabled.

More information can be found in Section 4.2, “Control Transfer.”

2.2 USB Endpoint Interrupt

If the endpoint successfully receives data (PID and CRC are correct), that data is then transferred to the buffer RAM, an ACK packet is generated, the packet size is reported to the UEPxDSR register, and the TRFC flag is set.

If the TCxIE flag is set then an endpoint interrupt is generated when the TRFC flag is set.

Within the USB_Enable() function the TCxIE flags are set to all active endpoints. The endpoint ISR is also added during compile time.

As EP0 cannot be disabled, bit TC0IE is set by default in the USB_Enable() function.

3 Descriptors

A USB device is described by a hierarchy of descriptors: type of device, class, power consumption, number of interfaces, endpoints, packet length, identification number of vendor and device, strings with readable information on the device, etc. A descriptor is a data structure with a defined format that is sent to the host during enumeration to let the host know everything about the device plugged into the USB port. There are three basic descriptor types: standard, vendor, and class specific descriptors. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor, followed by a byte-wide field that identifies the descriptor type. Chapter 9.5 of the *Universal Serial Bus Rev. 2.0 Specification* contains a discussion of descriptors.

Each device must provide some standard descriptors — device, configuration, interface, and endpoint. String descriptors are not mandatory. Standard descriptor structures are defined in the usb.h file.

According to the *Universal Serial Bus Rev. 2.0 Specification*, each USB function can have only one device descriptor, a few configuration and interface descriptors with alternative settings, and endpoints; however, the MC68HC908JW32 USB module allows only one configuration, two interfaces with no alternate settings, and a maximum four endpoints.

Memory organization is important. PCs use little endian memories, however the Freescale 8-bit MCU uses big endian. This must be considered if you create descriptor data structures in C language. Change byte order of word time items in a structure. You can use the CHNG_ENDIAN(x) macro for the 16-bit long types, and CHNG_ENDIANL(x) for the 32-bit long types. Also, be careful parsing standard requests. Macros are stored in the usb.h file.

Example of device, configuration, interface, HID, and endpoint descriptors of a three button mouse:

```
const DeviceDscStrc device01 = { // device descriptor
  sizeof(DeviceDscStrc), // length of this descriptor
  DSC_DEVICE_TYPE,     // this descriptor is DEVICE
  CHNG_ENDIAN(0x0200), //USB release num. in BCD 02.00 - word must change endian
  0x00, // device Class Code CLASS_CDC_DEVICE
  0x00, // device subclass code
  0x00, // device protocol code
  EP0_BUFFER_SIZE, // max. endpoint 0 buffer size -> always 8 see hw. spec
  CHNG_ENDIAN(0x5538), // vendor id. - Freescale - again endian must be changed
  CHNG_ENDIAN(0x1000), // product id
  CHNG_ENDIAN(0x0000),// device release number
  0x01,       // index of manufacturer string descriptor
  0x02,       // index of product string descriptor
  0x00,       // index of serial number string descriptor - zero - doesn’t exist
  0x01,       // number of possible configurations - always 1
 };
```

```
const Config01_dsc_strc config01 = { // configuration descriptor
   sizeof(ConfigDscStrc),    // length of descriptor
   DSC_CONFIGURATION_TYPE,   // type of descriptor - configuration
   CHNG_ENDIAN(sizeof(Config01_dsc_strc)), // length of config + interface +
        // + HID + endpoint descriptors - all are transferred together
   0x01, // number of interfaces used in function - at least one
   0x01, // value to use as an argument to the SetConfiguration() request
   0x00, // index of string descriptor describing this configuration
   0xA0, // bits: 7-reserve(one), 6-self powered, 5-remote wakeup, 4-0-reserved
   0x32, // // max. power consumption in this configuration 2mA unit -> 100mA

  sizeof(IntfcDscStrc), // interface descriptor length
   DSC_INTERFACE_TYPE, // type of descriptor
   0, // number of this interface.
   0, // value of alternate setting for the interface - must be zero - not allowed
   1, // number of endpoints used by this interface
   3, // class code HID CLASS
   0,  // subclass code
   2, // protocol code - MOUSE
   0, // Index of string descriptor describing this interface

  sizeof(HidDscStrc), // length of this descriptor
   DSC_TYPE_HID, // HID descriptor type
   CHNG_ENDIAN(0x0110), // HID Class Specification release number
   0, // Hardware target country
   1, // Number of HID class descriptors to follow
   DSC_TYPE_REPORT, // Report descriptor type
   CHNG_ENDIAN(sizeof(ReportDscStrc)), // Total length of Report descriptor
   // ENDPOINT 01 IN descriptor interrupt
  sizeof(EpDscStrc), // length od endpoint descriptor
    DSC_ENDPOINT_TYPE, // descriptor type identifier
    EP01_IN_ADR, // address of endpoint - defined by number and direction!
    (EP1_MODE>>6), // endpoint mode - use statement defined in usb_periph_cfg.h
    CHNG_ENDIAN(EP1_BUFFER_SIZE), // buffer size - use statement from
                                  //usb_periph_cfg.h
    0x0a, // interval for polling interrupt endpoint for data transfers
  };
```

4 Endpoint Transfer Types

As discussed, USB is a host-centric bus. The host initiates all the transactions. Each transaction is build up from three packets: the token, data, and handshake packets. The MC68HC908JW32 USB module takes care of communication. You do not need to know anything about PIDs, addresses, or endpoints. You only provide the USB module data to send and get the received data. The handshake packet is also generated by the USB module. The *Universal Serial Bus Rev. 2.0 Specification* discusses the packets in more detail.

By setting the endpoint type, the pipe made up from the endpoint uses one of the protocols defined in *Universal Serial Bus Rev. 2.0 Specification* — control, bulk, interrupt, or isochronous transfer type. The MC68HC908JW32 has implemented control, bulk, and interrupt endpoints.

4.1 Packet Types

4.1.1 Token Packet

The token packet starts all communications on the USB. It is issued by the host and contains the packet identifier (PID), device address (ADDR), and endpoint number (ENDP) to identify a consignee.

The PID carries information on the type of token. These are:

- IN token — the host wants to receive data in the next packet
- OUT token — the host wants to send data to a device in the next packet
- SETUP token — to start control transfer
- SOF token — the start of a frame

CRC covers address and endpoint fields and is checked by hardware.

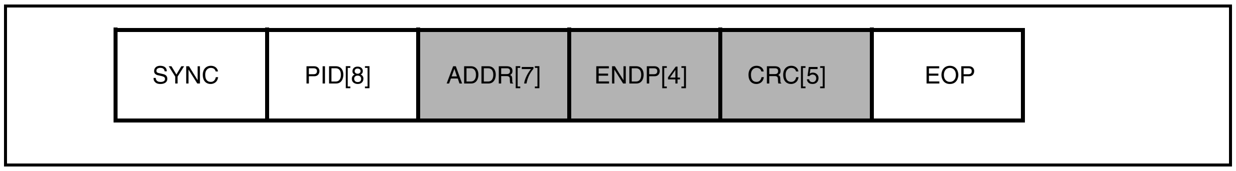

Figure 6. Token Packet Fields

4.1.2 Data Packet

A data packet is issued either by a device or by the host in accordance with the previous token packet. It contains PID and data parts. Types of PID are either Data1 or Data2 to distinguish odd and even data packets that are generated or checked by hardware. CRC protects all the data fields and is generated or checked by hardware also.

Maximum data payload for full-speed devices is 64 bytes.

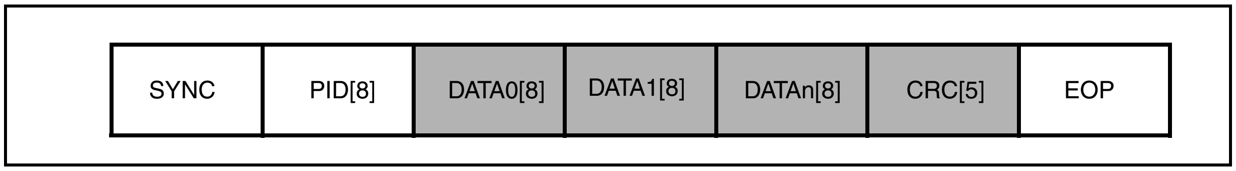

Figure 7. Data Packet Fields

4.1.3 Handshake Packet

Handshake packets report the status of a data transaction and can return values indicating the reception or rejection of data, halt conditions, or flow control. They are built only by PID.

There are three possible handshake packets:

- ACK — acknowledgment that the packet was successfully received
- NAK (Not Acknowledgment) — temporarily unable to send/receive data; there is no data to send concerning the interrupt transfer
- STALL — the endpoint is halted or a control pipe request is not supported, intervention from the host is required.

A handshake packet is issued or checked by hardware. If data is received correctly, the USB module issues an ACK. If the endpoint is busy and is not able to receive a data packet, or has nothing to send, hardware issues a NAK. If you halt the endpoint, hardware issues a STALL. If there is token or data error, hardware does not issue a response.

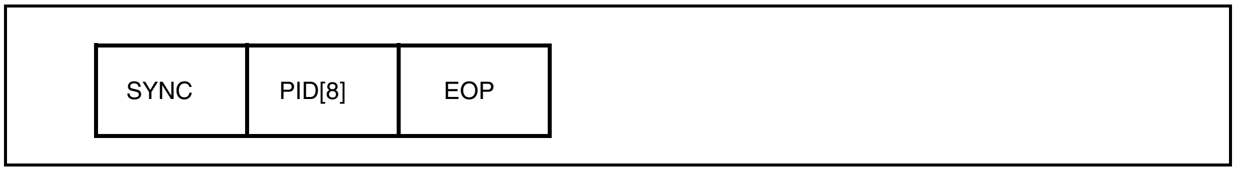

Figure 8. Handshake Packet Fields

4.1.4 SOF Packet

The start of frame transaction (SOF) packet consists of a PID indicating the packet type followed by an 11-bit frame number field. The packet is issued by the host at a nominal rate of once every 1.00 ms.

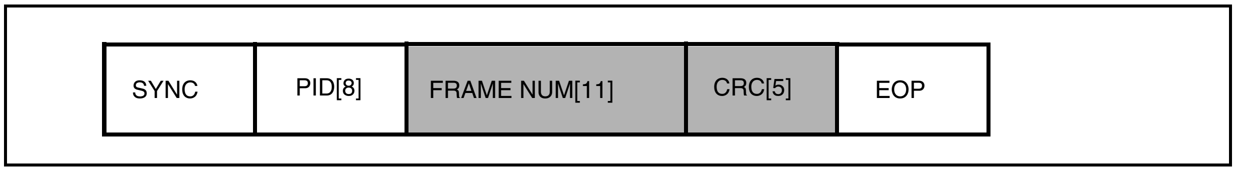

Figure 9. SOF Packet Fields

Table 3. Table of Possible PID

Packet type	PID name	Value
HANDSHAKE	ACK	0010B
	NAK	1010B
	STALL	1110B
DATA	DATA1	0011B
	DATA2	1011B
TOKEN	OUT	0001B
	IN	1001B
	SOF	0101B
	SETUP	1101B

4.2 Control Transfer

Control transfer is typically used to send different requests to a function: standard requests, vendor or device specific requests such as get descriptor, set address, set configuration, and so on. On the MC68HC908JW32, the control transfer is applicable only on the control pipe built by EP0 and where the packet is a maximum eight bytes long.

Control transfer consists of three stages:

- Setup stage that consists of token packet, data packet and handshake packet
- Multiple data stages in which one data stage consists of a token, data and handshake packet, and the last data stage has a data packet shorter than the maximal packet length
- Status stage confirms whether the previous setup and data stage finished successfully (see Figure 10 and Figure 11)

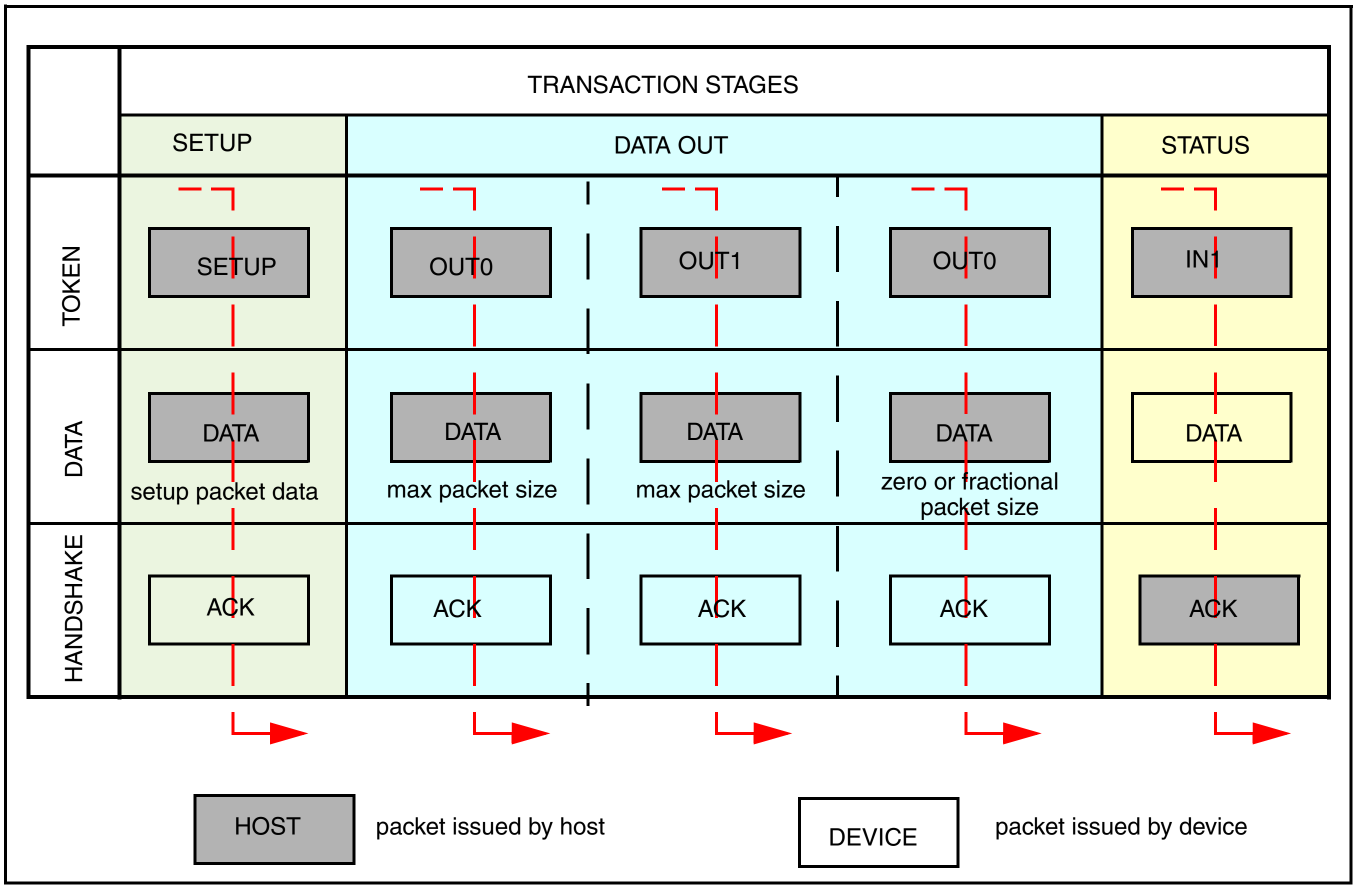

Figure 10. Control OUT Transaction — All Packets

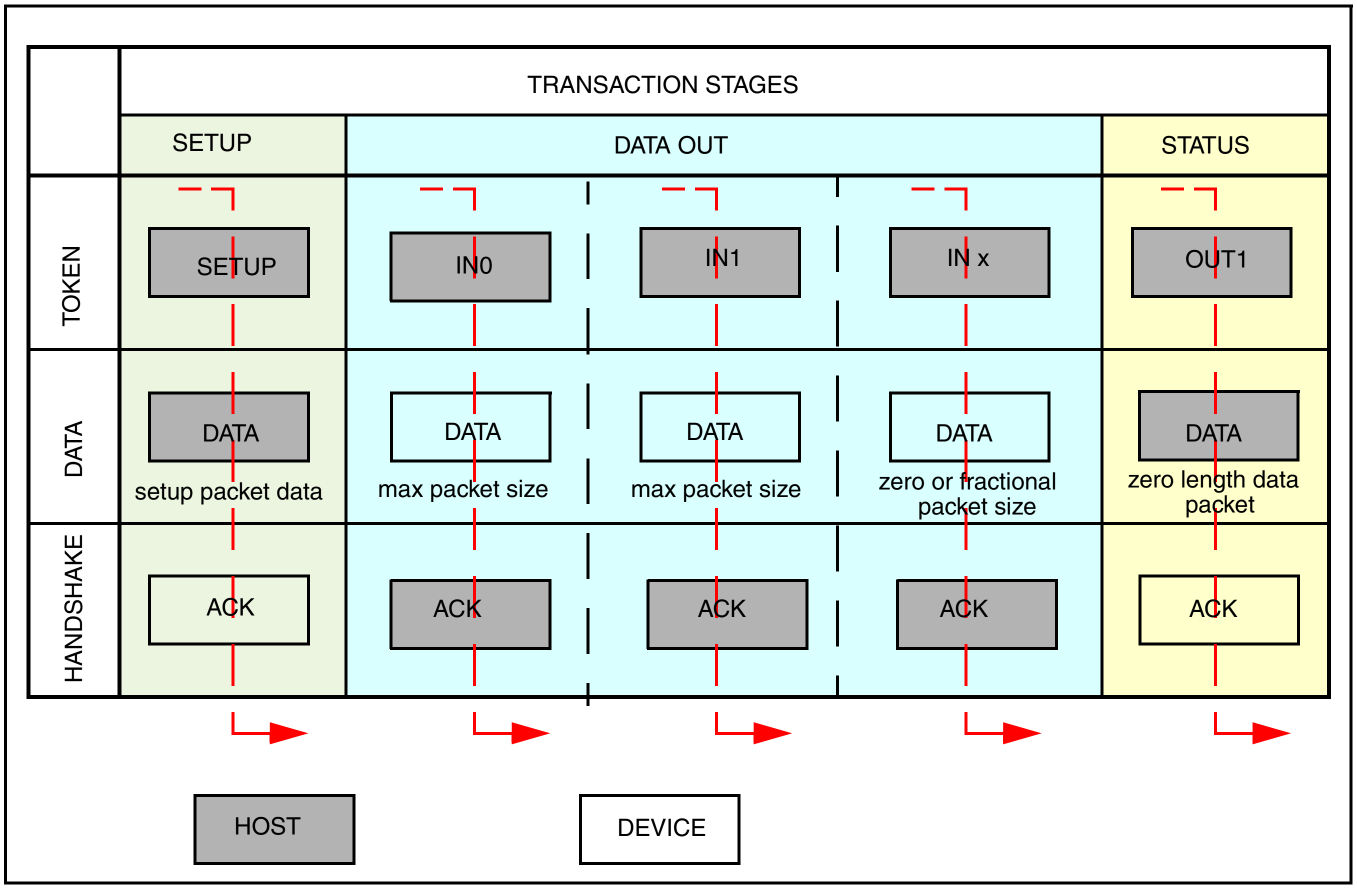

Figure 11. Control IN Transaction — All Packets

4.2.1 Setup Stage

The host initiates the control transfer by issuing two packets in the setup stage. The token packet carries the device address, endpoint number, and PID. The data packet has a setup packet format, which details the type of request. The setup packet structure SetupPcktStrc is defined in file usb.h. Chapter 9.3 of the *Universal Serial Bus Rev. 2.0 Specification* contains a more detailed description.

If the function receives setup data successfully, it responds with an ACK handshake packet. Handshake packets are generated by hardware and you need not give it any attention. If the function does not receive setup data, it does not send a handshake packet. Functions cannot issue a STALL or NAK packet in response to a setup request. CRC and PID checking are performed by hardware and user attention is not needed.

If CRC and PID are correct, you are notified by bits DVALID_IN, TRFC_IN and data is copied to the EP0 buffer.

Hardware also recognizes the token's PID as setup and generates the system interrupt — SETUP interrupt. You must parse the received data and prepare the USB module for the data stage.

If the data stage that follows is IN, you must prepare data to send. This is a typical situation if a GET_DESCRIPTOR standard request is received. You must fill your buffer with data and call the function:

```
uchar USB_TxBuff0(uchar* adr, uchar cnt)
```

adr is your buffer address and cnt is the number of bytes to send. The function returns number of bytes pending to transmit.

If the data stage that follows is OUT, you should prepare your buffer to receive data during the endpoint interrupt using the function:

```
void USB_RxBuff0(uchar* adr, uchar cnt)
```

adr is your buffer address , *cnt* is the number of bytes to receive.

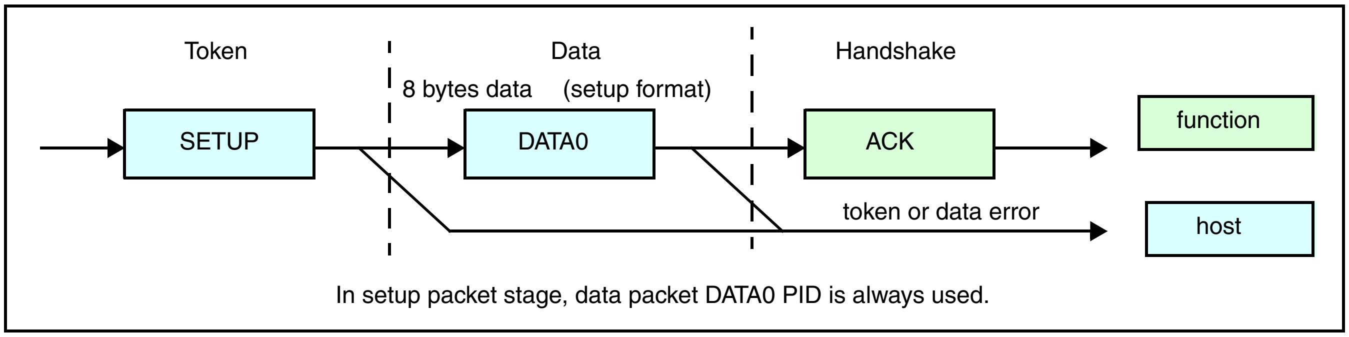

Figure 12. Control Transfer SETUP Stage

4.2.2 Data Stage (Optional)

If present, the data stage of a control transfer consists of one or more IN or OUT transactions and follows the same protocol rules as bulk transfer. All transactions in the data stage must be in the same direction, such as all INs or all OUTs. The amount of data to be sent during the data stage and its direction are specified during the setup stage. If the amount of data exceeds the maximal data packet size, the data is sent in multiple transactions (INs or OUTs) that carry the maximum packet size. Any remaining data is sent as a residual in the last transaction.

As mentioned above, the data stage can be IN or OUT.

IN: When the host is ready to receive control data from the function, it issues an IN token. If the function successfully receives the IN token it can either send data, send a STALL to indicate that the request is not supported, or send a NAK packet. A NAK packet indicates that function is either processing the last transaction or has no data to send. If an IN token has an error, the token is ignored. A multiple data stage is finished by sending a data packet shorter than the maximal packet size or a zero packet size.

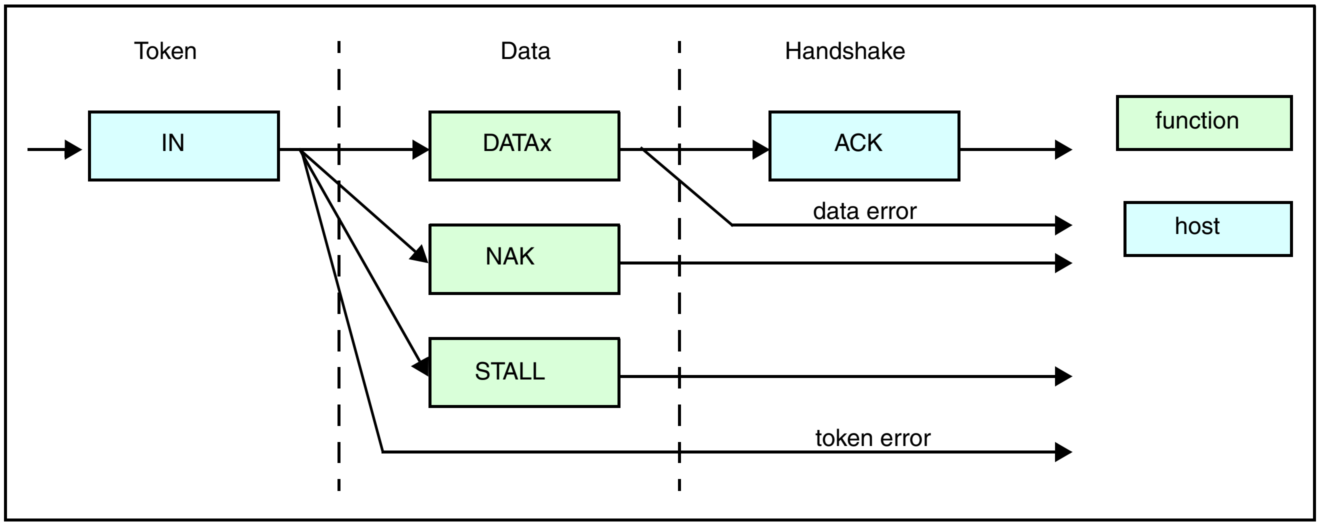

Figure 13. CONTROL Transfer — IN Data Stage

OUT: When the host wants to send control data to the function, it issues an OUT token and then a data packet. If the function receives a token and data successfully, it responds to the host with an ACK. If the function is processing previous data and the buffer is not empty, the function cannot receive any data and responds with a NAK to the host. A STALL is issued by the function to let the host know that the endpoint has an error and has halted. A multiple data stage is finished by sending a data packet shorter than the maximal packet size or a zero packet size.

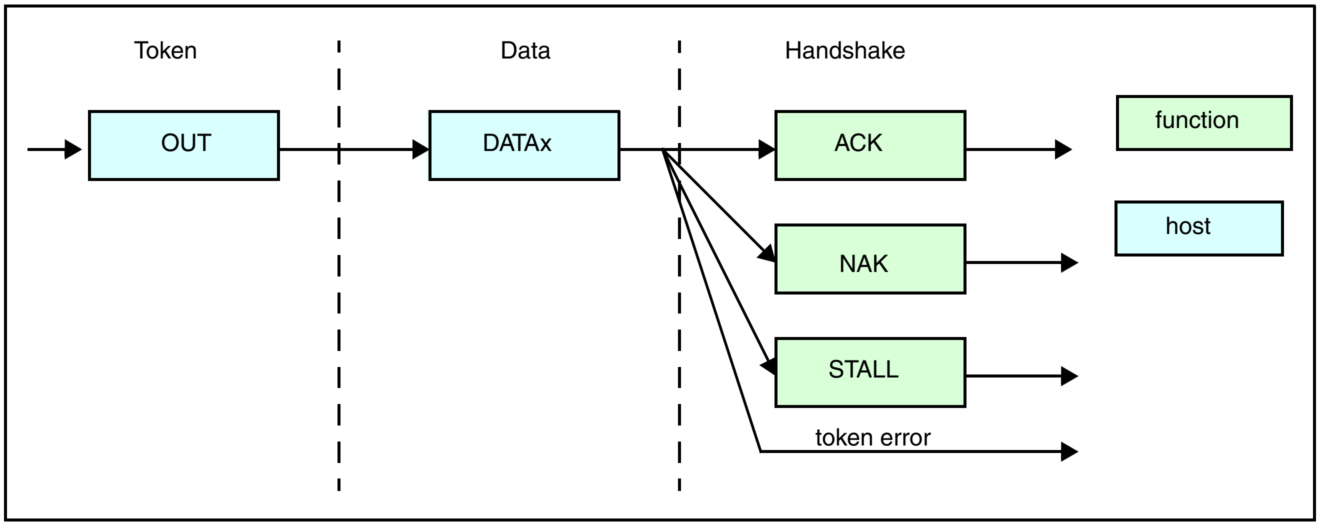

Figure 14. CONTROL Transfer — OUT Data Stage

4.2.3 Status Stage

The status stage reports to the host the outcome of the previous setup and data stages of the transfer. Three possible results may be returned:

- The transfer sequence completed successfully

- The transfer failed
- The function has an error and is busy

Unlike a handshake packet, you must prepare the status stage handshake after the last data packet is received or sent. The status stage packet can be received or transmitted using the functions:

```
uchar USB_TxBuff0(uchar* adr, uchar cnt)
uchar USB_RxBuff0(uchar* adr, uchar cnt)
```

The status stage is different for the IN/OUT data stage:

OUT: If the previous data stage was IN and logical data flow was from the function to the host, the host must acknowledge successful receipt of this data by sending an OUT token followed by a zero length data packet. The function responds with either

- An ACK — the function has completed the command and is ready to accept a new command
- A NAK — the function is processing the command and that the host should continue the status stage
- A STALL — the function has an error that prevents it from completing the command.

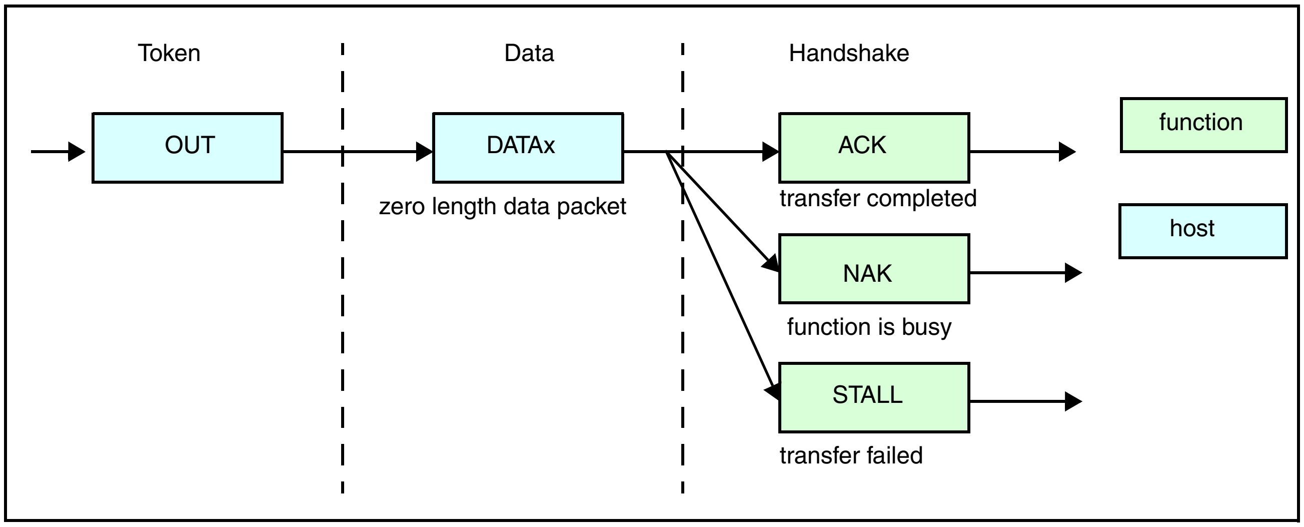

Figure 15. CONTROL Transfer — OUT Status Stage

IN: If the previous data stage was OUT and logical data flow was from the host to the function, the function must acknowledge successful receipt of this data by sending a zero length data packet as a response to the IN token. The USB driver uses the function USB_RxBuff0(DUMMY_ADR, 0) to send a zero data packet as a response to the IN token.

If the function is processing data, it can issue a NAK and the host will retry the status stage later. If the function detects any error during the token or data stages it responds with a STALL.

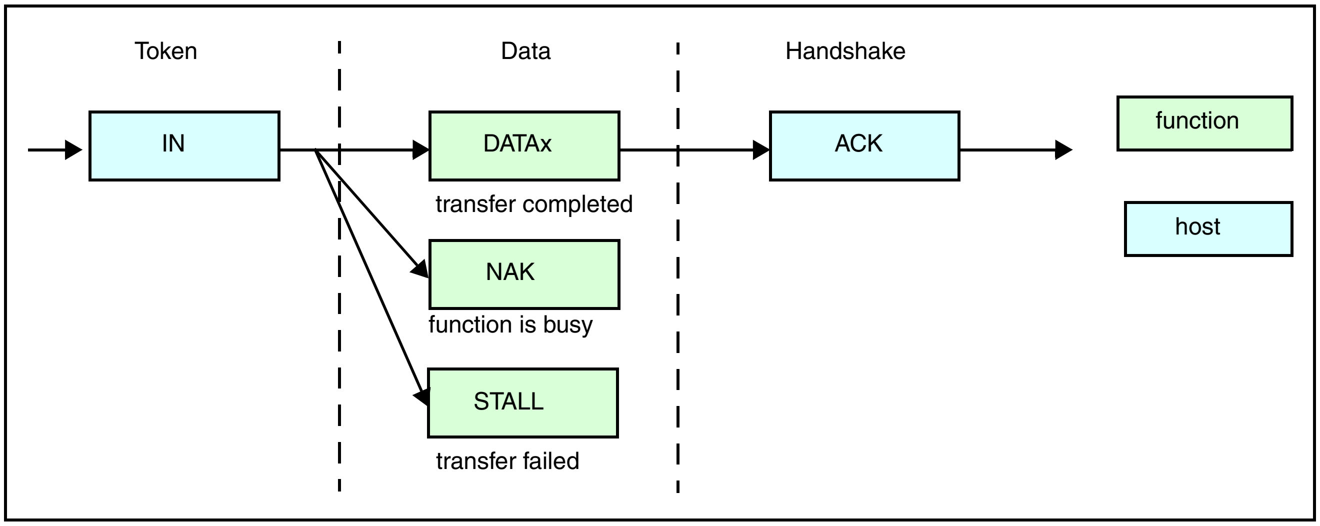

Figure 16. CONTROL Transfer — IN Status Stage

4.2.4 Programming Control Transfer

As discussed in Section 4.2, “Control Transfer,” control transfer is mainly used to send different types of requests to the function. To simplify firmware, the MC68HC908JW32 has an integrated request processor in the USB module. The request processor automatically processes some standard USB requests without firmware attention (see MC68HC908JW32 Data Sheet table 11-2).

After the setup stage containing a SETUP token is successfully received, the MC68HC908JW32 request processor automatically processes some USB requests such as SET_ADDRESS and CLEAR_FEATURE. Only if GET_DESCRIPTOR or SYNC_FRAME standard requests, vendor, or class specific requests are received, is the status stage data packet loaded into the EP0 buffer and a SETUP interrupt generated.

The USB driver software parses the setup stage data packets and switches according to type of request, whilst also preparing the GET_DESCRIPTOR (DEVICE, CONFIGURATION and STRING) standard request response (see Figure 17).

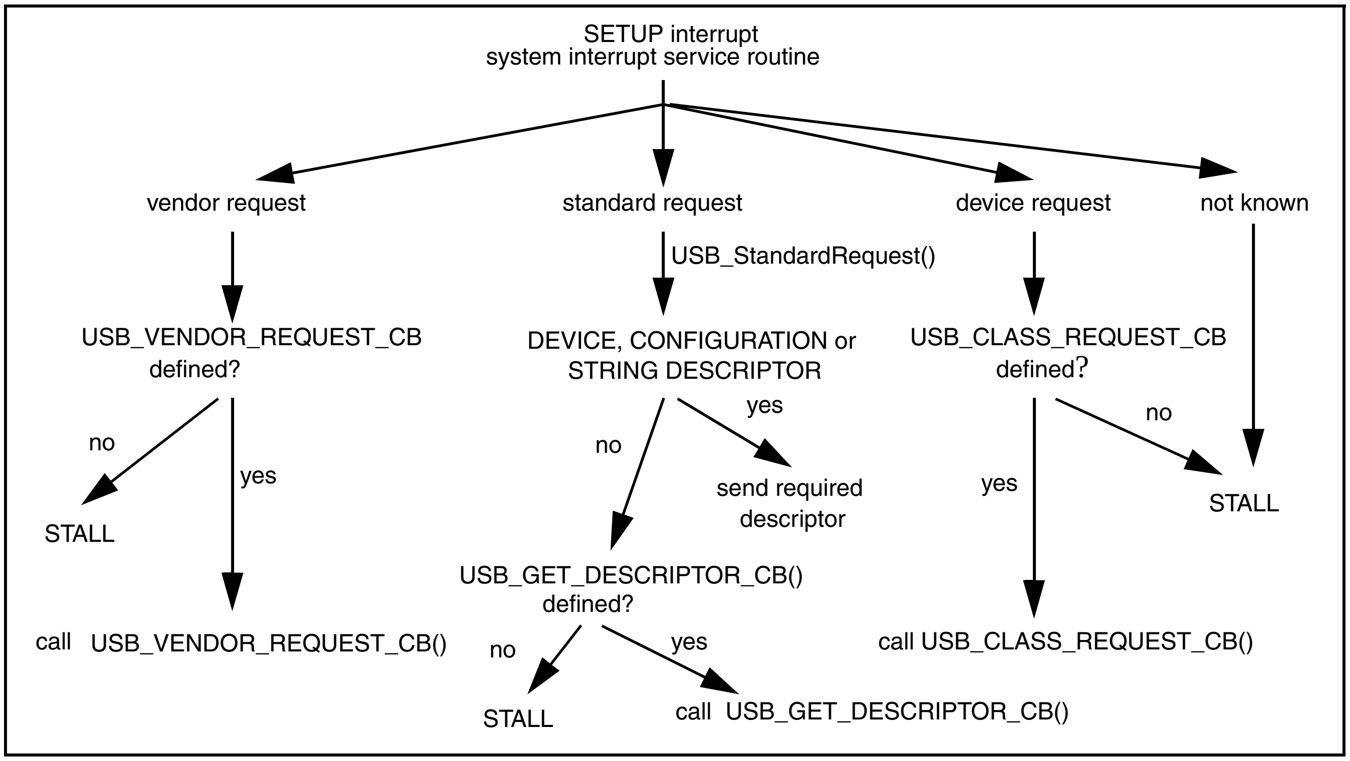
SETUP interrupt
system interrupt service routine
vendor request
standard request
device request
not known
USB_StandardRequest()
USB_VENDOR_REQUEST_CB defined?
DEVICE, CONFIGURATION or STRING DESCRIPTOR
USB_CLASS_REQUEST_CB defined?
no
yes
STALL
send required descriptor
USB_GET_DESCRIPTOR_CB() defined?
call USB_VENDOR_REQUEST_CB()
call USB_CLASS_REQUEST_CB()
call USB_GET_DESCRIPTOR_CB()

Figure 17. Parsing SETUP Stage Data Packet

If the request is a vendor type, then the function USB_VENDOR_REQUEST() is called, if defined. If you want to process any vendor requests you must write your own function USB_VendorRequest() function and define the macro constant

```
#define USB_VENDOR_REQUEST_CB  USB_VendorRequestCB
```

in file usb_periph_cfg.h to let the USB driver know what function to call. If the statement USB_VENDOR_REQUEST_CB is not defined as the function then vendor requests are responded with STALL.

Class requests are processed in the same way. If you want to process any class request, you must create the function USB_ClassRequest_CB() and define macro the constant

```
#define USB_CLASS_REQUEST_CB  USB_ClassRequestCB
```

in file usb_periph_cfg.h to let the USB driver use it. The standard requests can be only a GET_DESCRIPTOR subtype, as the USB module request processor responds to all the others. If a standard request is received then the USB_StandardRequest(void) function is called.

The function parses the GET_DESCRIPTOR request and if the DEVICE, CONFIGURATION or STRING descriptor is requested, USB firmware prepares the response. Otherwise, the USB_GET_DESCRIPTOR_CB() function is called, if defined.

To let the USB driver know where the descriptors are stored, define a pointer to the descriptors variable. There are macro statements prepared to create pointers to descriptors in flash memory (usb.h file).

Use the macro statements

```
IDENT_CONFIG_DSC(config01);
IDENT_DEVICE_DSC(device01);
```

where config01 and device01 are the configuration and device descriptors structures.

These macros define the variables ptrToConfigDsc and ptrToDeviceDsc in Flash, and initialize them by the descriptors address. These variables are used in the USB driver software.

There is usually more than one string descriptor. Pointers to string descriptors must be stored in a table, which is an array of pointers to the string descriptors. The length of the string table must be defined in the usb_periph_cfg.h file as a macro constant

```
#define STRING_DSC_TAB_LEN   4
```

(there are four string descriptors). Another macro constant must name the string table, to let the USB driver know where the table lies

```
#define STRING_DSC_TAB  stringDscTab
```

If a GET_DESCRIPTOR command requires a descriptor other than a DEVICE, CONFIG or STRING, then you must a prepare response in the function USB_GetDescriptor_CB() and define macro constant

```
#define USB_GET_DESCRIPTOR_CB  USB_GetDescriptorCB
```

to let the USB driver call the callback function. If the request is unknown, then a STALL is issued.

The data stage is managed by the USB driver through code in the EP0 ISR and a pre-negotiated amount of data is transferred or received. When a data packet is shorter than the maximal packet length is transferred, the data stage is finished.

The status stage is also managed by the USB driver, so no user attention in needed.

4.3 Bulk Transfer

A bulk transaction type is characterized by the ability to guarantee error-free delivery of data between the host and the function by means of error detection and retry. Bulk transactions use a three-packet transaction consisting of token, data, and handshake packets as shown in Figure 18 and Figure 19.

OUT: If the host wants to send bulk data to the function, it issues an OUT token and then issues the data. If either the token or the data is corrupted, the function does not respond. If data is successfully received in the buffer, the function responds with an ACK handshake. If the endpoint buffer is full and the function is temporarily not able to receive data, it issues a NAK to let the host know it should try to send the bulk data later. If the function has a permanent error, it issues a STALL.

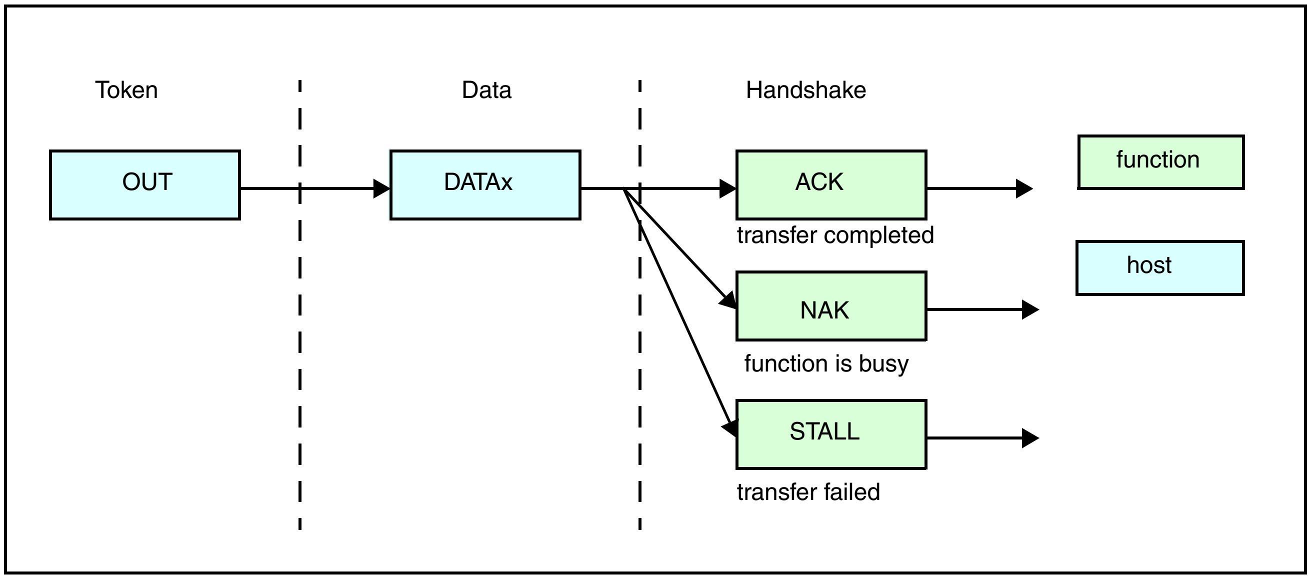

Figure 18. Bulk/Interrupt OUT Transfer — All Packets

IN: If the host is ready to receive data from a function, it issues an IN token. If the function is ready, it sends a data packet. If the function has a permanent error, it issues a STALL. If the function has nothing to send, it issues an NAK.

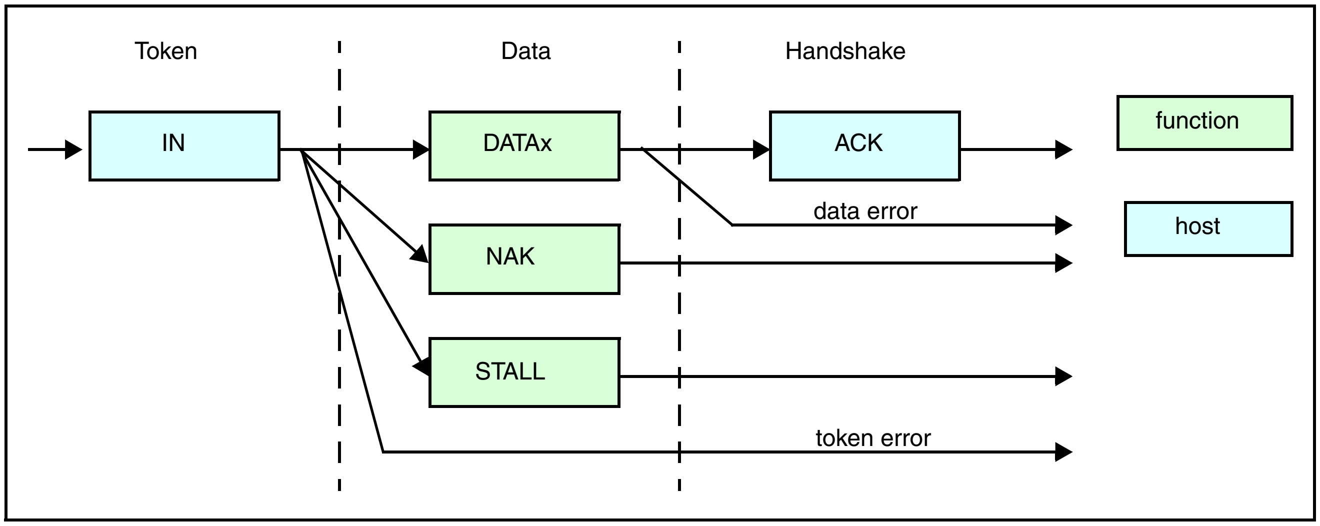

Figure 19. Bulk/Interrupt Transfer IN — All Packets

4.4 Interrupt Transfer

Unlike bulk transfer, interrupt transfer assures guaranteed communication latency. If the function has data to send, it fills the buffer and waits for an IN data token which is issued in a pre-negotiated period by the endpoint descriptor.

Interrupt transactions may consist of IN or OUT transfers.

Upon receipt of an IN token, a function may return data, a NAK, or STALL. If the endpoint has no new data to return, the function returns a NAK handshake during the data phase. If the halt feature is set for the interrupt endpoint, the function will return a STALL handshake. If an interrupt is pending, the function returns the interrupt information as a data packet. In response to receipt of the data packet, the host issues either an ACK handshake if data was received error-free, or returns no handshake if the data packet was received corrupted.

If an OUT token packet followed by interrupt data packet is issued by the host, for example if the host wants to send interrupt data to the function, the function must respond with an ACK packet if the data was successfully received in the endpoint buffer; with a NAK if the endpoint buffer is full and the function is not able to receive other data; or with a STALL if the endpoint is halted. If an OUT token data is corrupted, the function does not respond.

4.4.1 Programming Bulk and Interrupt Transfer

Both bulk and interrupt transfers use the same functions to send or receive data. The only difference is the latency guarantee with interrupt transfer. The USB driver provides a few functions to send and receive data over USB. You can use either buffer transfer to send or receive data or a character approach — to send or receive characters.

Receive (OUT Direction):

For the buffer receive use the function

```
uchar USB_RxBuffx(uchar* adr, uchar cnt)
```

To receive data by the OUT endpoint, define your own buffer where the received bytes will be stored. The size of the buffer is arbitrary.

Parameters adr contains the start address of the user buffer and cnt passes the number of bytes to receive. When the function is called, it immediately copies any previously received data from the endpoint buffer to the user buffer. If the function copies all data from an endpoint buffer so that the endpoint buffer is empty, bit UEPxCSR[DVALID] is cleared to enable reception of the next data packet and the function ends.

The function returns remaining space in the user buffer in bytes. The function is non-blocking and the remaining requested bytes are transferred to the user buffer within the next endpoint complete transfer ISRs.

When the next data packet has been received by the endpoint buffer, an ACK handshake is issued by hardware and the interrupt service routine USB_EP_ISR is called. If there is any remaining space in the user buffer, the endpoint buffer copies the received data to the user buffer. If the endpoint buffer becomes empty, bit UEPxCSR[DVALID] is cleared to enable reception of other data.

When some bytes remain in the endpoint buffer because the user buffer is full, (UEPxCSR[DVALID] = 1), then the next OUT data packet cannot be received and a NAK handshake packet is issued by hardware.

Another OUT data packet can be received after the endpoint buffer is empty.

You may also use the function

```
uchar USB_RxBuffPendingx(void)
```

which returns the number of bytes required to receive in the user buffer

```
uchar USB_GetRxReadyx(void)
```

which returns the number of data bytes pending in the endpoint buffer

Another possibility to receive data is “byte by byte”. You can use the blocking function

```
uchar USB_RxCharx(void)
```

The function waits until a previously called reception has finished (reception initiated by the USB_RxBuffx() function), and until there is at least one byte in the endpoint buffer (the function is blocking). Then the function returns the received byte and ends. If the endpoint buffer is empty and the function reads out the last pending byte, the function clears bit UEPxCSR[DVALID] to enable the next data reception.

It is useful to out if there is any byte pending in the endpoint buffer. The function uchar USB_GetRxEmptyx(void) returns the number of received bytes pending in the endpoint buffer.

Transmit (IN direction):

There are two possibilities for transmitting data over a USB with a blocking and a non-blocking function. Using the function uchar USB_TxBuffx(uchar* adr, uchar cnt) you can send a buffer. You must create the user buffer and fill it with data to transmit. Variable *adr* is the pointer to the user buffer and *cnt* passes the number of bytes to transmit.

The function returns the number of bytes pending in the user buffer for transmission.

When the function is called, it starts to copy data from the user buffer to the endpoint buffer. When all data is copied or the endpoint buffer is full, bit UEPxCSR[DVALID] is set, the buffer is primed to transmit data, and data is sent as soon as a data IN token is received. When the buffer has been successfully sent to the host and an ACK handshake received, an interrupt service routine (USB_EP_ISR) is called. If there is any data pending in the user buffer, it is copied to the endpoint buffer which is primed again to send it. The procedure is repeated until there are no bytes pending in the user buffer.

If there is no data to send (the endpoint buffer is not primed), the module responds with a NAK to all IN data tokens.

Function uchar USB_TxBuffPendingx(void) returns the number of bytes pending for transmission in the user buffer.

Function uchar USB_GetTxEmptyx(void) returns number of bytes available in the endpoint buffer.

A byte by byte approach should be used with the function uchar USB_TxCharx(uchar ch). In this function, ch contains the byte to be sent, which is copied to the endpoint buffer. When the endpoint buffer is full, bit UEPxCSR[DVALID] is set to arm the buffer. If you want to send a packet shorter than the endpoint buffer size he should flush the buffer with the function uchar USB_TxFlushx(void). The function returns the number of sent bytes.

If you want to halt the endpoint, you may call the macro statement USB_WRSTALL_EPx(x) which will set/clear bit UEPxCSR[STALL].

5 Enumeration

Enumeration is the process in which the host determines the function parameter, such as the number of the endpoint, configuration, and class. The enumeration process on a standard Windows PC is:

1. Plug the function into the attached USB hub. The function is powered up and pulls the D+ line up via USB_Init(). The pullup must be enabled in USB_Init().
2. Host detects the D+ line and waits a further 100 ms to let the function finish the insertion process. During this time, you must initialize the MCU's PLL, USB module (USB_Enable()).
3. Host issues a reset state to the USB line. After reset, the function is in the default state.
4. Host sends a GET_DESCRIPTOR (device descriptor) request. When the first eight bytes of the device descriptor is received by the host, it issues a reset state.
5. Host issues a SET ADDRESS request, the function goes to the addressed state.
6. Host sends a GET_DESCRIPTOR (device descriptor) request again for all 18 bytes.
7. Host sends a GET_DESCRIPTOR (configuration descriptor) request of nine required bytes. Function returns nine bytes. The first nine bytes of the configuration descriptor contain information on the full size of the configuration descriptor.
8. Host sends a GET_DESCRIPTOR (configuration descriptor) and asks for 255 bytes so the function returns the whole configuration descriptor.
9. If there are any string descriptors defined, the host asks for them.
10. Host sends a SET_CONIGURATION request. If the host responds correctly, the function is configured and can be used. The host polls all working endpoints to allow transfer data.

6 HID Class—Mouse

One of the most simple USB devices is the human interface device (HID) such as a mouse or keyboard. This example describes a simple PC mouse. Windows will recognize a standard USB device and install it to Windows as a human interface device. You can check this in the device manager.

In Figure 5, device 1 represents a mouse. It consists of the EP0 control pipe and EP1— interrupt endpoint.

Control pipe EP0 is used during the enumeration process to send descriptors to the host. It uses control transfer.

In addition to the standard descriptors, such as device descriptor, configuration descriptor, interface and endpoint descriptor, the HID class devices also reports an HID and report descriptor. If the mouse wants to tell the computer that it has a pressed button or was moved, it sends a report. The structure of the report is flexible and it is configured in a report descriptor. As the USB_StandardRequest() function answers only standard descriptors, there is a possibility of answering all the non-standard and device-specific descriptors within the function USB_GET_DESCRIPTOR_CB(). It is your responsibility to write the USB_GET_DESCRIPTOR_CB() function. Chapter 6.2.2 of the *USB Device Class Definition for Human Interface Devices (HID)* contains a detailed description of this process.

The report descriptor below describes a simple mouse with a 4-byte report. The first byte is defined on lines 6 to 18 of the report descriptor and reports the mouse buttons. Within the first byte there are three defined bits — report count and report size on lines 13 and 14. The next five bits in lines 16 and 17 are not used here and are padding.

The axis of movement of the mouse are defined on lines 20, 21, and 22. Report values must lie in the interval –127 to 127 and are defined on lines 25 and 26 as three time 8-bytes.

```
1  const ReportDscStrc reportDsc =
2   {
3      0x0501,              // Usage Page(Generic Desktop)
4      0x0902,              // Usage (Mouse)
5      0xA101,              // Collection(Application)
6      0x0901,              // Usage(Pointer)
7      0xA100,              // Collection(Physical)
8      0x0509,              // Usage Page(Buttons)
9      0x1901,              // Usage Min(01)
10     0x2903,              // Usage Max(03)
11     0x1500,              // Logical Min(0)
12     0x2501,              // Logical Max(1)
13     0x9503,              // Report Count(3)
14     0x7501,              // Report Size(1)
15     0x8102,              // Input(Data,Variable,Absolute)
16     0x9501,              // Report Count(1)
17     0x7505,              // Report Size(5)
18     0x8101,              // Input(Constant)- Padding
19     0x0501,              // Usage Page(Generic Desktop)
20     0x0930,              // Usage(X)
21     0x0931,              // Usage(Y)
22     0x0938,              // Usage(WHEEL)
23     0x1581,              // Logical Min (-127)
24     0x257F,              // Logical Max(127)
25     0x7508,              // Report Size(8)
26     0x9503,              // Report Count(3)
27     0x8106,              // Input(Data,Variable,Relative)
28     0xC0,                // End Collection
29     0xC0,                // End Collection
30   };
```

For the report descriptor, the report structure looks like that below: variable but represents buttons press/release, x/y represent mouse movement <-127, 127>, and w represents wheel rotations <-127, 127>.

```
typedef struct
 {
   byte  but;           // buttons (bits 0, 1, 2)
   byte  x;             // mouse movement X axis
   byte  y;             // mouse movement X axis
   byte  w;             // mouse wheel
 }MouseReportStrc;
```

HID class functions should also respond to a few class specific requests such as GET_REPORT, SET_PROTOCOL, and GET_PROTOCOL. The function USB_CLASS_REQUEST_CB() must respond to these standard requests.

EP1 sends reports to the host. EP1 is an interrupt endpoint so the host periodically polls the function through the IN data token. If the function detects any change such as a button press or movement, it should respond with a report data packet, otherwise the IN data token is NAK.

7 Communication Device Class (CDC)

In the past, the serial line known as UART was the major way of connecting embedded devices to a PC. Many new computers do not have COM ports, so USB is a good substitute. In situations when a PC application does not see the difference between a standard and the Virtual COM port, you can easily migrate from UART to USB by using the virtual serial COM port on the PC. There is also the standard Microsoft driver usbser.sys. This class is called abstract control model serial emulation and is described in Chapter 3.6.2.1 in *Universal Serial Bus Class Definitions for Communication Devices*.

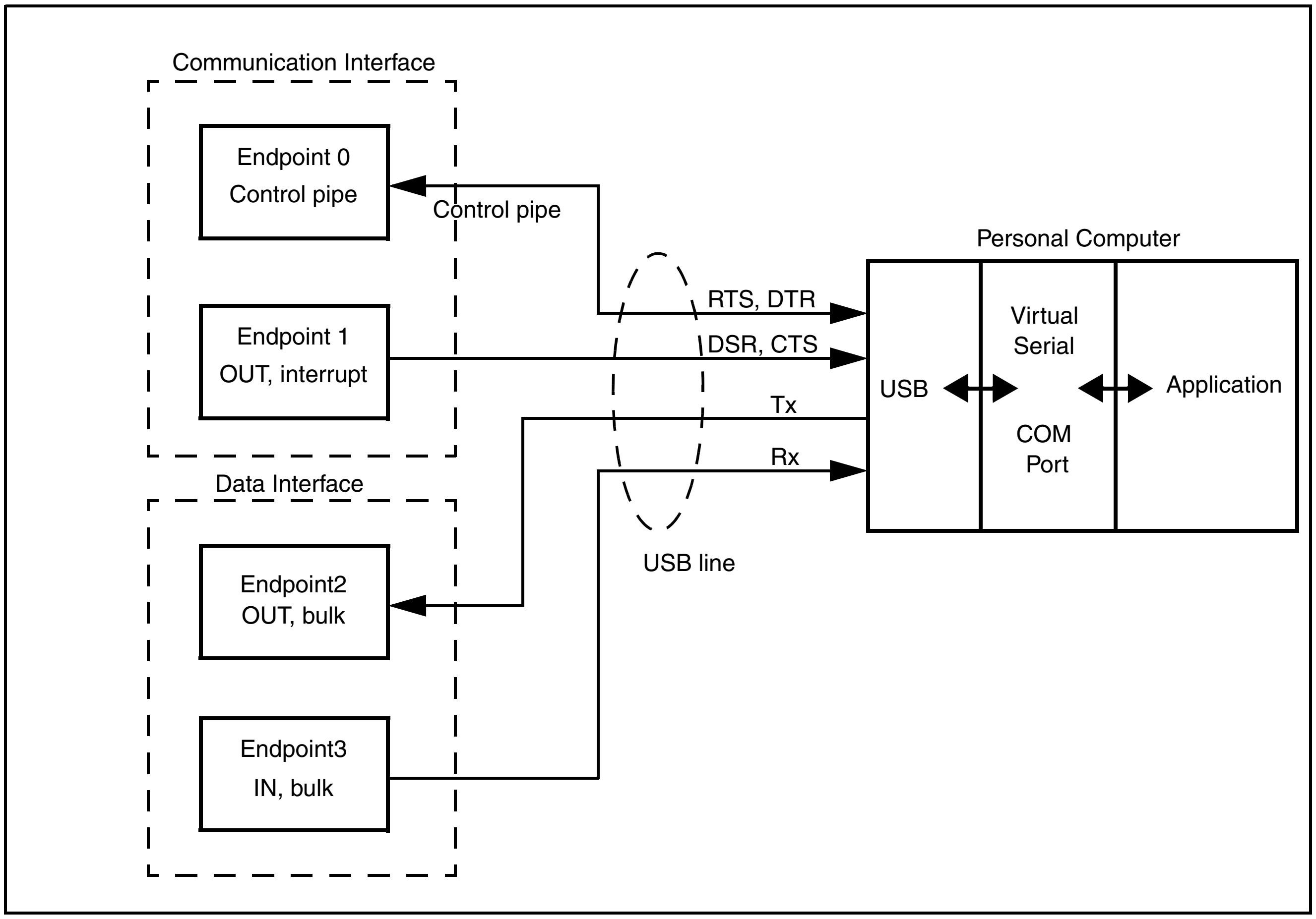

Figure 20. Virtual Serial COM Port

The virtual serial COM port is made up of two interfaces: the data class interface and the communication class interface. The communication class interface consists of at least two pipes: control pipe (EP0) for management, and OUT interrupt EP1 pipe for notification.

The control pipe is used for standard commands, such as GET_DESCRIPTOR and SET_ADDRESS, and class specific commands such as SEND_ENCAPSULATED_COMMAND, GET_ENCAPSULATED_RESPONSE (required), SET_CONTROL_LINE_STATE, GET_LINE_CODING, and SET_LINE_CODING.

SEND_ENCAPSULATED_COMMAND and GET_ENCAPSULATED_RESPONSE are device-specific commands that are used to send the AT commands over the USB. Implementation of those requests is required but we do not work with them when received.

SET_LINE_CODING and GET_LINE_CODING are device-specific requests that are transferred when a COM port on the PC is opened. In fact, this is only information on how the COM port is set and does not affect the speed or the setting of the USB communication. When a COM port is opened on the PC, the host asks the device how it is set with the request GET_LINE_CODING. When you change the baudrate, parity, or number of data bits for example, in the hyperterminal, the host sends the settings to the device with the request SET_LINE_CODING. The data packet has three bytes, with the structure below.

Structure of a line coding data packet

```
typedef struct
{
   unsigned long   dwDTERate;    // data terminal rate, in bits per second
   byte    bCharFormat;          // stop bits:
                                 //    0 : 1 Stop bit
                                 //    1 : 1.5 Stop bits
                                 //    2 : 2 Stop bits

   byte    bParityType;                 // parity:
                                 //   0 : None
                                 //   1 : Odd
                                 //   2 : Even
                                 //   3 : Mark
                                 //   4 : Space
   byte    bDataBits;                // data bits (5, 6, 7, 8 or 16)
 } CdcLineCodingStrc;
```

The SET_CONTROL_LINE_STATE request is used to send the state of the RTS and DTR signals to the device. A data packet has 16 bits. Bit zero represents signal DTR. Bit 1 is RTS.

The second pipe in the communication class interface is the notification pipe. This is used to notify the host of the state of the device lines. It is not required, but this pipe should be an interrupt. Serial state notification should look like:

Table 4. Serial State Notification

bmRequestType	bNotification	wValue	wIndex	wLength	Data
10100001B	SERIAL_STATE	Zero	Interface	2	UART State

Where SERIAL_STATE has value 0x20.

UART state is a 16-bit variable:

D15..D7 Reserved for future use.

D6 Received data has been discarded due to overrun in the device.

D5 A parity error has occurred.

D4 A framing error has occurred.

D3 State of the device ring detection signal.
D2 State of the device break detection signal.
D1 State of the device transmission carrier signal. This signal corresponds to RS-232 signal DSR.
D0 State of the device receiver carrier signal. This signal corresponds to RS-232 signal DCD.

Data interface has two endpoints: IN and OUT are both are bulk.

You can use function USB_TxBuffx(uchar* adr, uchar cnt) to send a buffer, or USB_TxCharx(uchar ch) to send data byte by byte. If you use the USB_TxCharx() function, you are filling the endpoint buffer which is send when full or you can send the buffer intermediately by USB_TxFlushx().

To receive the data, you may use USB_RxBuffx(uchar* adr, uchar cnt) to receive a defined amount of bytes, or USB_RxCharx() to pull out one byte from the endpoint buffer (if there is a received byte).

Code example (byte approach):

```
// RS 232 function
for(;;) {
   if(USB_GetRxReady2() > 0) // if there is any received byte
       receiveChar = USB_RxChar2(); // move it from endpoint 2 buffer to “receiveChar“

   if(USB_GetTxEmpty3()) { // if there is at least one free byte available in endpoint 3 buffer
      (void)USB_TxChar3(transmitChar); // write “transmitChar” byte to endpoint 3 buffer
      (void)USB_TxFlush3(); // let the USB module send endpoint 3 buffer to USB line
   }
}
```

Document Number: AN3153
Rev. 0
08/2006