Freescale Semiconductor
Application Note

Document Number: AN3560
Rev. 1, 07/2008

USB Device Development with the MC9S08JM60

In-depth Understanding of the MC9S08JM60 USB Module

by: Derek Liu
Asia & Pacific Operation Microcontroller Division

1 Introduction

MC9S08JM60 devices form part of the Freescale Flexis series of microcontrollers. The Flexis^{TM} series of microcontrollers is the connection point of the Freescale Controller Continuum where 8-bit and 32-bit compatibility becomes reality.

The 8-bit MC9S08JM60 MCUs are devices with a full-speed USB module — providing best-in-class USB module performance, system integration, and software support. The USB module of the JM series MCU has seven endpoints and 256 bytes RAM for high efficiency data transfer.

MC9S08JM60 MCUs provide many peripherals, such as USB, SPI, IIC, SCI, ADC, TPM, and RTC. These MCUs are flexible and can easily be integrated into different applications such as game pads, security control panels, printers, and PC peripherals.

Contents

More detailed information on the USB module of MC9S08JM60 devices and how to use it in applications are discussed in this document. In addition, some skills and issues that must be noticed in design have also been discussed.

2 MC9S08JM60 USB Module Introduction

Figure 1 shows the block diagram of the USB module for the MC9S08JM60 MCU. It includes:

- On-chip 3.3 V regulator (VREG)
- USB transceiver (XCVR)
- Serial interface engine (SIE)
- USB RAM

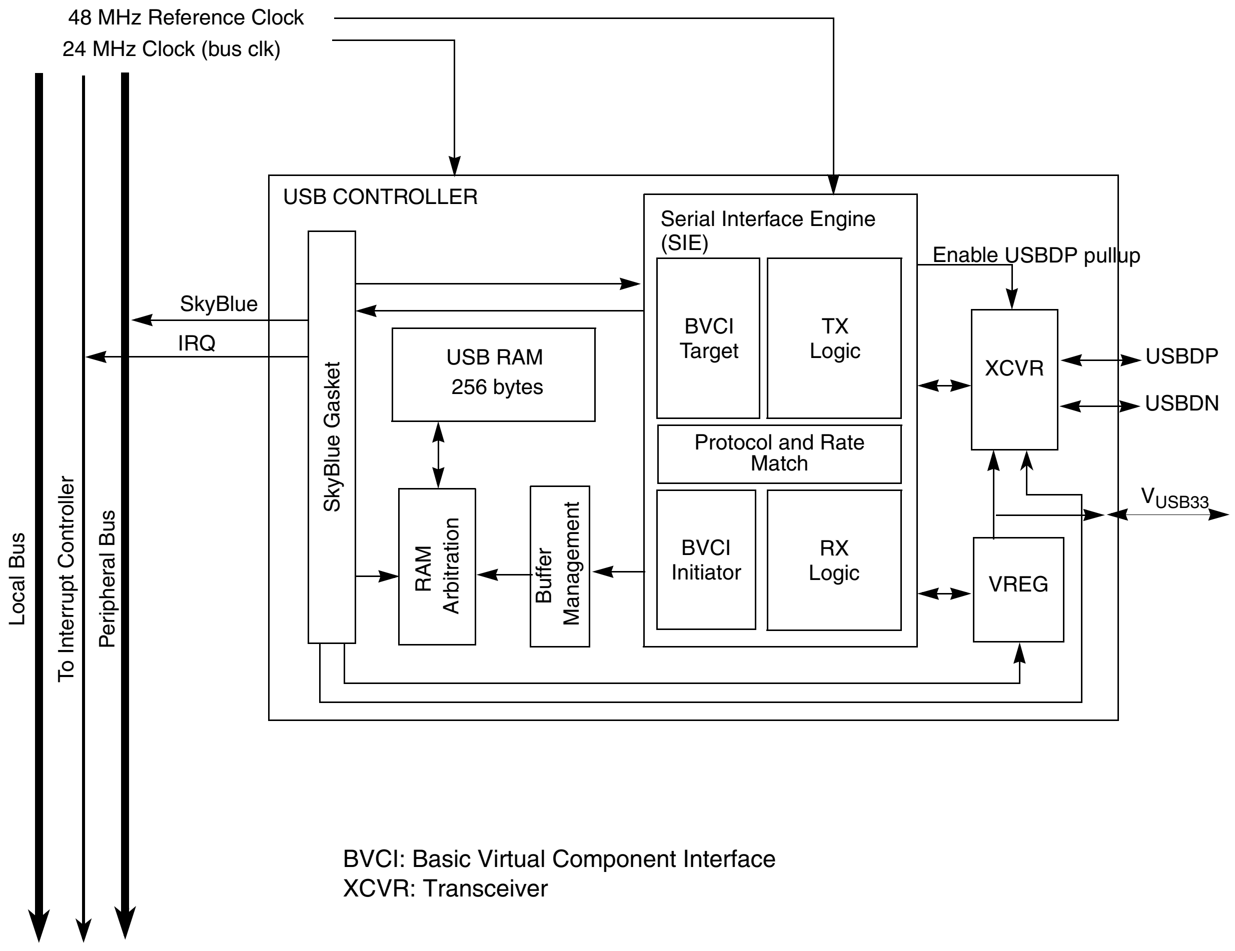


Figure 1. USB Module Block Diagram

Figure 2 shows the layers of a typical USB device.

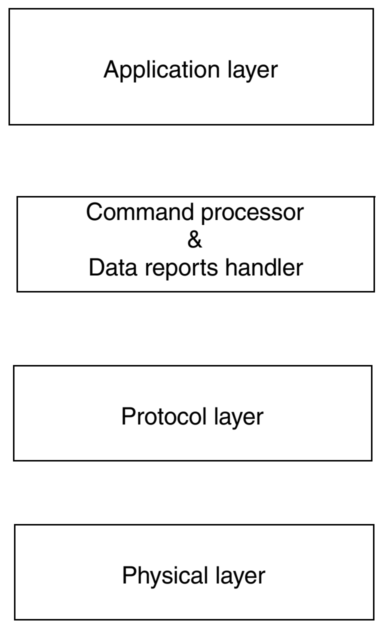


Figure 2. USB Protocol Layer

The physical layer involves data signaling, such as J and K signals, start of packet (SOP), and resume signals. The MC9S08JM60 USB PHY corresponds to the physical layer.

The protocol layer lies above the physical layer. Transactions of a different transfer type are managed by this layer. The packet in the USB low-level protocol are processed in this layer. In the MC9S08JM60 USB module they are finished by a SIE.

The command processor and data report handler layer lies between the protocol and application layers. This layer corresponds to the USB stack firmware described in this document. The USB enumeration and configuration, the standard class request, or the customized protocol are all processed by this layer.

The top layer is the application layer. It it is based on the USB communication provided in the lower layers. The program in this layer focuses on the customized application.

2.1 USB Endpoint

One important concept needs to be discussed before explaining the USB module — the endpoint. Each USB logical device consists of a collection of independent endpoints. An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. A USB pipe is an association between an endpoint on a device and software on a host.

The MC9S08JM60 device has seven endpoints that can be used to build seven communication pipes between the USB host and device.

Endpoint 0 is bidirectional (in and out), mainly used for control transfer. Endpoint 0 is required for all USB devices.

Endpoint 1–6 are directional, they can be configured to do only in or out direction communication at a time.

Endpoints 5 and 6 are double buffering (also called ping-pong buffering). Each of these has two buffers. One buffer can be operated by the CPU, when the other communicates with the host (controlled by SIE). These two buffers exchange their roles after the current transaction is over. The CPU has the control of one of the double buffers in turn. With this feature, the communication efficiency improves because the CPU waiting time is shortened.

Each endpoint can be enabled or disabled by the EPCTLn register. The direction and handshake for all endpoints are also controlled by the EPCTLn register.

The control transfer, bulk transfer, isochronous transfer, and interrupt transfer are all supported by each endpoint of MC9S08JM60 series MCU. That matches all kinds of USB data transfer requirements.

2.2 VREG

On-chip regulator (VREG) provides a stable power source to the USB internal transceiver and internal (or external) pullup resistor. It requires a voltage supply input in the range of 3.9 V to 5.5 V, and its output voltage range is from 3.0 V to 3.6 V. The VREG shares the same power supply with the MCU, but the MCU can work with the power voltage from 2.7 V to 5.5 V.

The corresponding pin to regulator output voltage is $V_{USB3.3}$, that can be used to supply the power for the external pullup resistor.

The on-chip regulator can be enabled or disabled by USBVREN bit of the USBCTL0 register. If it is disabled, an external 3.3 V voltage must be provided to USB module via $V_{USB3.3}$ pin.

NOTE

Damage to the part may occur if the external 3.3 V power supply is provided while the internal regulator is enabled.

2.3 USB Transceiver

USB transceiver belongs to the physical layer of the USB standard protocol. It provides a differential signal for USB communication. The USBDP and USBDN pins are analog input/output lines for full-speed data communication.

2.4 USB SIE

The USB serial interface engine (SIE) is mainly used to implement transferring and receiving of logic.

For the SIE transferring logic, the firmware immediately needs to configure three BD registers and fills the data into endpoint buffer; then hands over the control to SIE. The SIE will send the data to the host automatically after the host issues an IN operation. All work required in the USB specification, such as coding (NRZI), bit stuffing, CRC, SYNC, etc., are implemented by SIE without firmware's intervention. The SIE hands over the control to the CPU after the transfer is finished.

As for the receiving logic, the SIE can receive data packets automatically. The receiving logic process includes decoding, synchronizing, detection, bit stuff removal, EOP detection, CRC validation, PID check, etc.

The data packet is filled into the endpoint buffer, and the BD registers are updated. Then the SIE gives the control to the CPU for reading data out. Some status and flag bits in the USB registers (INSTAT, STAT, ERRSTAT, and EPCTLn) for this transaction are refreshed. The SIE owns the control of the BD and endpoint buffer before the data ends to receive.

The SIE sends the acknowledge (ACK) or non-acknowledge (NAK) handshake to the host. It also stalls the current transfer (STALL). The handshake is enabled or disabled by the EPHSHK bit in the EPCTLn register. If it is enabled, ACK is transferred to the host when the transaction is correctly received. The NAK is sent out by SIE if the OWN bit in BD status and control register is 0. The STALL is issued by setting the EPSTALL bit in the EPCTL register or the BDTSTALL bit in the BD status and control register.

With the SIE transferring and receiving logic, the work of the firmware is greatly reduced resulting in more CPU time being saved.

2.5 USB RAM

MC9S08JM60 MCU provides 256 bytes RAM space for the USB module, in which the USB BD registers and endpoint buffer are located. The USB RAM can also be allocated by the system as general purpose RAM when the USB module is disabled or it still has free space.

The USB RAM runs at 48 MHz, which is twice the speed of the bus clock. It can be accessed by the CPU system and SIE. This is an area to exchange data and information between the CPU and SIE.

All the USB control and the status registers (BD registers) and USB data buffer must be allocated in the USB RAM. It is illegal to allocate them in other space.

The USB RAM occupies a separate space instead of a part of the MCU RAM. From the memory map of MC9S08JM60 in Figure 3, the USB RAM address range is from 0x1860 to 0x195F, but the 4K bytes of MCU RAM starts from 0x00B0, ends at 0x10AF.

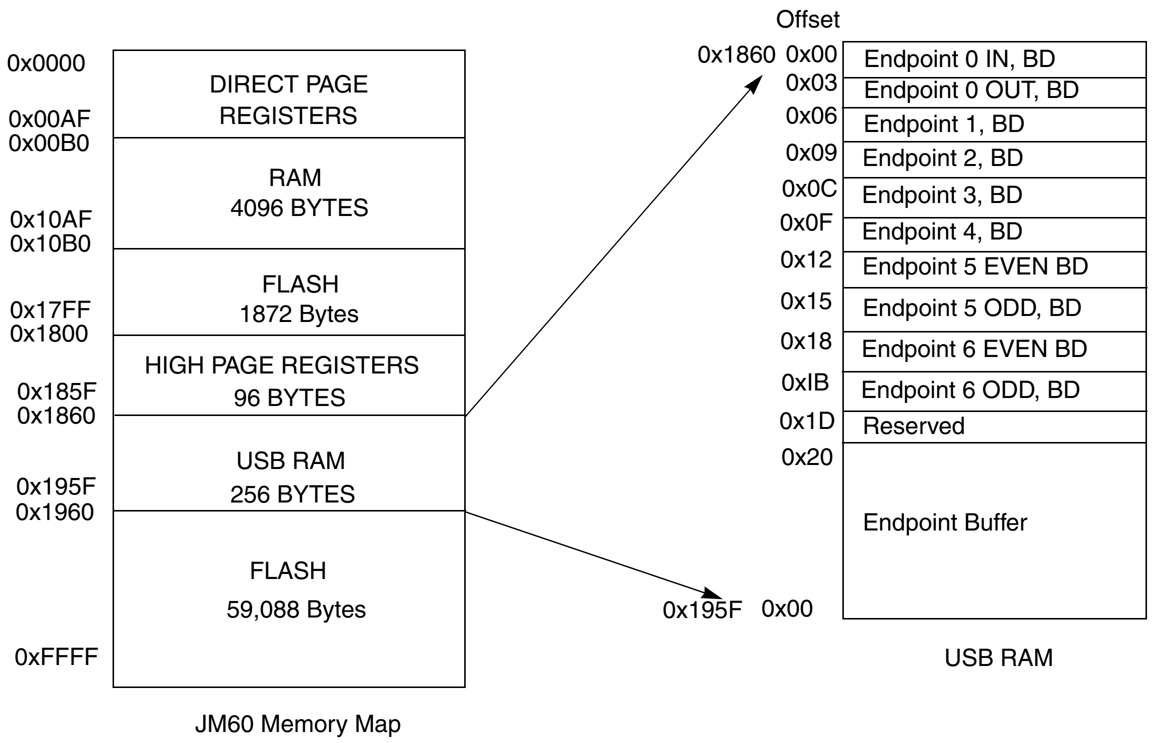


Figure 3. USB RAM

Figure 3 shows the USB RAM location of MC9S08JM60 in the memory map. The organization of USB RAM is also illustrated.

2.5.1 Buffer Descriptor Table (BDT)

Buffer descriptor table (BDT) is used to manage USB endpoint communication efficiently. It provides the status and control for each endpoint.

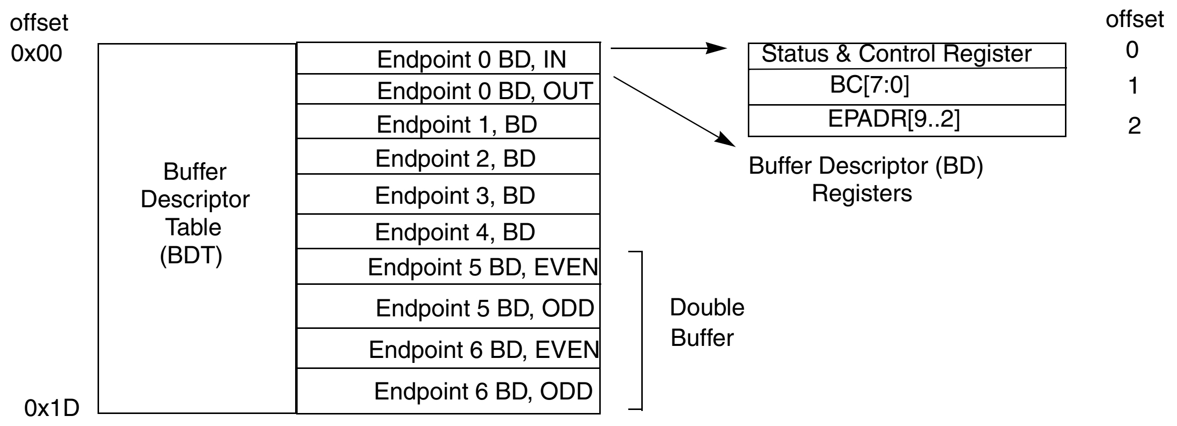


Figure 4. Buffer Descriptor

These registers for each endpoint are called buffer descriptor (BD). Each BD comprises three bytes, in which the endpoint data buffer's location, length, control logic, and status are kept.

The BDT locates at the beginning of USB RAM. The BDs for seven endpoints are organized and placed one by one from EP0 in to EP6 out. There are 10 BDs in total and 30 bytes are allocated from 0x1860 to 0x187D (offset: 0x00–0x1D in USB RAM space).

Figure 4 shows the BDs for seven endpoints and three BD registers for endpoint 0 (in direction). For endpoint 0 (bidirectional), 5 and 6 (double buffer), each has two BDs. Endpoint of 1–4 owns only one BD.

The first byte of the BD is the status and control register (see Figure 5). It includes four bits that the CPU reads or writes for controlling the USB communication. They are OWN (bit 7), DATA0/1(bit 6), DTS (bit 3), and BDTSTALL (bit 2).

Bit	7	6	5	4	3	2	1	0
	OWN	DATA0/1	0 BDTKPID[3]	0 BDTKPID[3]	DTS BDTKPID[1]	BDTSTALL BDTKPID[0]	0	0

Figure 5. Status and Control Register of BD

- BDTSTALL can be set by the CPU to stall the next transaction. A STALL handshake is issued on its associated endpoint if the BDTSTALL bit is set.
- The DTS bit is used to enable or disable the data toggle synchronization.
- DATA0/1 bit defines a DATA0 or DATA1 packet to be transmitted or received.
- OWN bit is implemented as the semaphore mechanism. Its value determines whether the BD and endpoint buffer is controlled by the CPU or SIE. When it is set to logic 0, the CPU has the control of the BD and endpoint buffer, the CPU can modify the contents of the BD registers and the data in the endpoint buffer. If it is set to logic 1, the SIE has the control of BD and endpoint buffer.
- The bit[5:2] are updated by SIE after a new token packet is received. The token packet PID is kept in these four bits.

The firmware designer must rewrite all bits in the status and control the register in terms of the configuration for the next transaction. Otherwise all the data packets may not be received or transferred. The wrong setting for DATA0/1 results in the data being discarded by the SIE or the host. The DTS and BDTSTALL bits can also result in the data packet not being received or being stalled. The values for these two bits are replaced by bit 0 and bit 1 of the new packet's PID. They must be updated before the next packet is allocated.

The second byte (BC[7:0]) is the length of the data packet to be transferred or the packet that has been received. It can be set to zero for some packets (e.g, the transaction in the status stage of control transfer). The maximum length is 64 for the MC9S08JM60 MCU. The BC register must be set to the actual packet size to be transferred for the in direction endpoint, but it must be initialized to be greater than or equal to the packet size that is received for out direction endpoint.

The third byte (EPADD[9:2]) keeps the start address of the endpoint buffer. It must be filled with the bit 9–2 of the relative address in USB RAM. It can be calculated by shifting two bits of the relative address to right so the endpoint buffer address is four bytes alignment.

Figure 6 shows an example for calculating the value of the EPADD register. Provided that the endpoint buffer locates 0x1880 (absolute address), the address relative to the start address of USB RAM is 0x20. Thus the value of EPADD for the endpoint can be calculated by shifting the value 0x20 two bits to right and the result is 0x08.

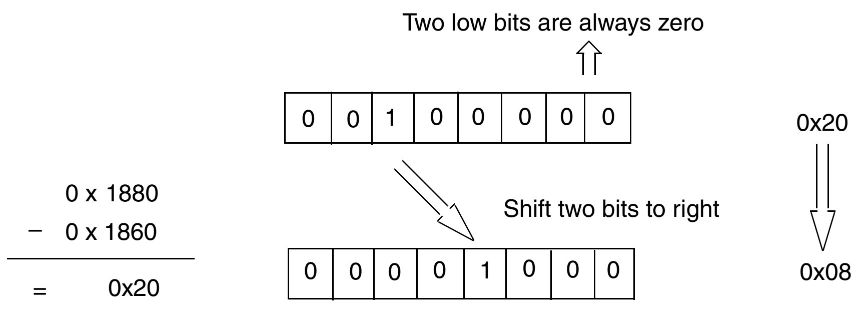


Figure 6. Example for Calculating the Value of EPADD Register

Setting these registers correctly is the basic requirement for successful endpoint communication. Figure 7 is an example of BDT configuration. It shows the content in the BD registers of the endpoints 0 and 1.

In this example, the buffer length is eight bytes for endpoint 0 (in and out) and four bytes for endpoint 1. The endpoint buffers are allocated continually from 0x1880 (relative address 0x20).

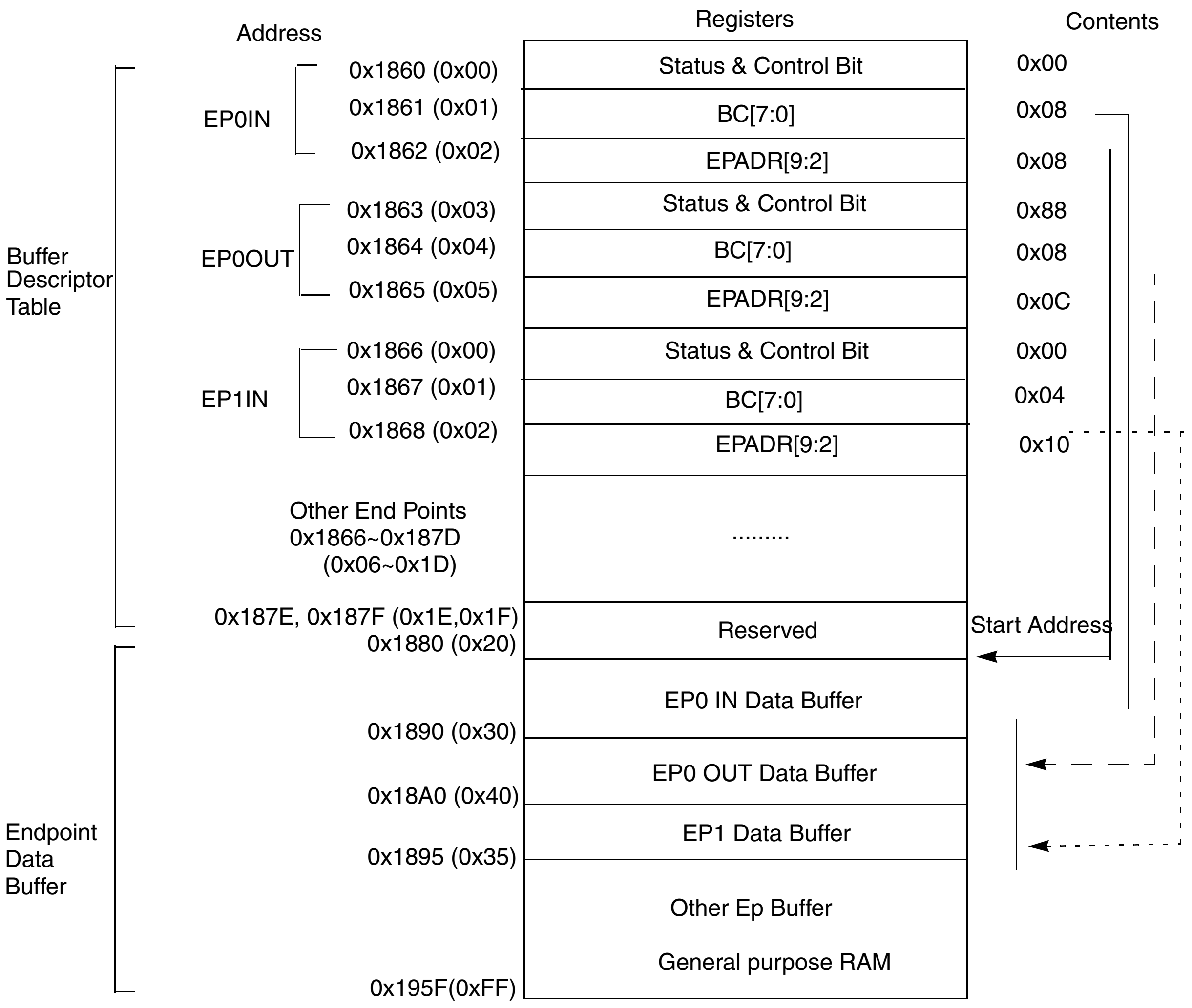


Figure 7. BDT Configuration Example

- Endpoint 0 in direction:

  If the content of the status and control register is 0x00, the CPU still has the control of BD.

  If the BC[7:0] register is filled with 0x08, it means that the packet size transmitted by EP0 in is eight bytes. This value takes effect when the SIE controls this endpoint.

  The maximum packet size for the endpoint is transferred to the host in the USB device enumeration process. The value for the BC registers can be less than the maximum value — the data that exceeds the maximum length must be divided into several packets. The actual number to be transferred to the host must be set to the BC register before the OWN bit of the status and control register is set (hand over the control of BD to SIE).

  The value of EPADD[9:2] is 0x08. It can be calculated by the start address allocated for endpoint 0 in direction. Its absolute address is 0x1880. The calculation method is illustrated in Figure 6.

- Endpoint 0 out direction:

  The BD for endpoint 0 out direction is set to receive a DATA0 packet. The content of the status and control register is 0x88. This means the control for BD and data buffer is handed over to SIE (OWN = 1). Then endpoint 0 receives DATA0 (DATA0/1 = 0) packet. When the data toggle has been enabled (DTS = 1), the correct data packet is received (STALL = 0).

  The SIE changes the OWN bit to 0 for handing over the control to the CPU. When one packet is finished receiving, the exact data length is updated and written to BC register.

- Endpoint 1 in direction:

  The endpoint 1 is used for the in direction (transmitting data to the host). It is configured to transfer only four bytes.

NOTE

When developing firmware, the programmer must avoid the address overlap for different endpoint buffers. In addition, the maximum size of the endpoint buffer must be the same as the length in the USB configuration descriptor.

The start address of the endpoint buffer must be sixteen bytes aligned in USB RAM.

As for the space in USB RAM that is not assigned, the firmware can use them for another purpose.

The flowchart in Figure 8 shows the operation of BD registers used for in and out direction. In out direction, after the data packet is received from the host, the TOKDNEF bit is set, and the CPU becomes the control of the BD and endpoint buffer (OWN = 0). Firmware can deal with this in an interrupt service routine or in a polling routine. The program reads the length of the received data from the BC register and then reads out the data from the endpoint buffer and processes it if necessary. After that, firmware updates the BD register and hands over the control to SIE for next transaction.

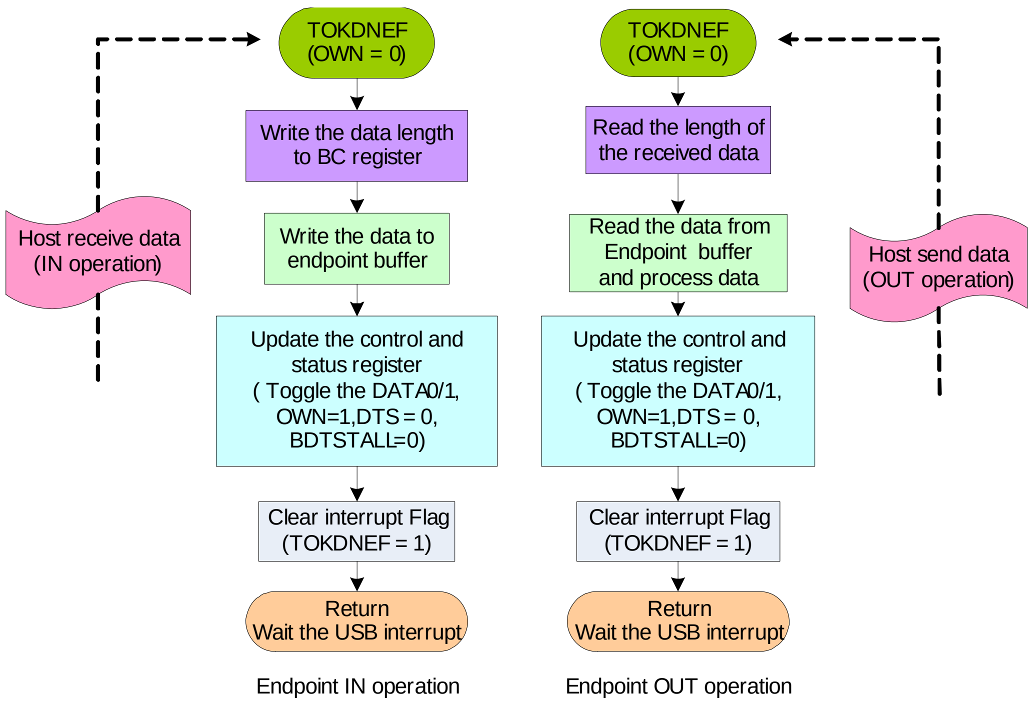

TOKDNEF
(OWN = 0)
Write the data length to BC register
Write the data to endpoint buffer
Update the control and status register (Toggle the DATA0/1, OWN=1,DTS = 0, BDTSTALL=0)
Clear interrupt Flag (TOKDNEF = 1)
Return Wait the USB interrupt
Host receive data (IN operation)
Endpoint IN operation
TOKDNEF
(OWN = 0)
Read the length of the received data
Read the data from Endpoint buffer and process data
Update the control and status register (Toggle the DATA0/1, OWN=1,DTS = 0, BDTSTALL=0)
Clear interrupt Flag (TOKDNEF = 1)
Return Wait the USB interrupt
Host send data (OUT operation)
Endpoint OUT operation

Figure 8. BDT Operation Example

Before the first packet is received, the firmware initializes the associated BD registers and hands over the control to SIE. Then the USB module is ready to receive data from the host.

In in direction of control transfer, the USB device receives one data packet that includes one request or command, then the device begins to prepare the data to be delivered to the host according to some protocols. The firmware writes the data length to the BC register, writes the data to the endpoint data buffer, then updates the status and control the register (DATA0/1, OWN, etc.). After that, the USB module begins to wait for the host to read data (in operation).

The TOKDNEF bit is set after the host finishes reading out the data in the endpoint (the transaction has finished). The USB interrupt is triggered if it is enabled. The new data for the next transaction can be prepared in an interrupt service routine or in a polling routine.

2.5.2 Double Buffer (Ping-Pong Buffer)

Endpoints 5 and 6 of the MC9S08JM60 MCU USB module are double-buffering endpoints (seeTable 1 on page 15). Either of them has two sets of BD (even and odd) registers. When one BD and its associated buffer are being operated by SIE at the same time; the CPU can access or control the other BD and attached data buffer.

The CPU and SIE exchange the control of the BD registers and endpoint buffer after they finish their work. The CPU gives the control to SIE as processing takes place for the data operation and the configuration of the BD registers. The SIE returns the control to the CPU and tries to control the other buffer when it finishes the current transaction. With the communications going on, the CPU and SIE have the control of two BDs and their attached data buffers in turn, so the double buffer is also called the ping-pong buffer.

The mechanism of the double buffer makes the CPU and SIE cooperate well without waiting for each other for a long time — hence the efficiency of communication is improved.

The flowchart in Figure 9 shows how to use the ping-pong buffer. Both BDs and their attached buffers are initialized at first. The double buffer has two sets of BD and data buffers that occupy different space in USB RAM. They both belong to one endpoint and have same direction (in or out) and share one EPCTLx(5 or 6) register.

The ODD bit in the STAT register reflects the operation completed by SIE. The user's program understands that BD and attached buffer is dealt with in terms of the value expressed in this bit. The ODD bit can be reset to point to even buffer by setting the OODRST bit in the CTL register.

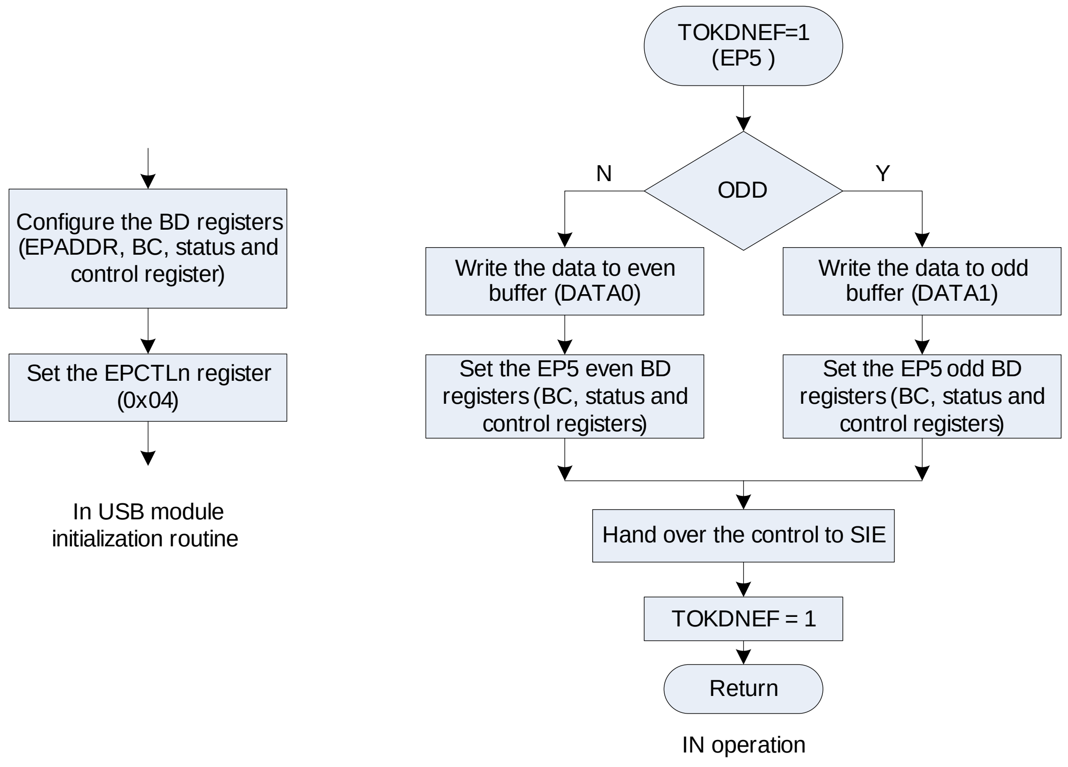

TOKDNEF=1
(EP5)
ODD
N
Y
Configure the BD registers (EPADDR, BC, status and control register)
Set the EPCTLn register (0x04)
In USB module initialization routine
Write the data to even buffer (DATA0)
Write the data to odd buffer (DATA1)
Set the EP5 even BD registers (BC, status and control registers)
Set the EP5 odd BD registers (BC, status and control registers)
Hand over the control to SIE
TOKDNEF = 1
Return
IN operation

Figure 9. Operation for Double Buffer (Ping-Pong Buffer)

The operation of a ping-pong buffer for in direction is illustrated in Figure 9. After the transaction with in operation ends, the TOKDNEF bit is set and the MCU becomes the controller of BD. Then the firmware reads the ODD bit to determine which BD and data buffer must be dealt. After that, the data buffer, the BC register, and the control register are written or updated. The MCU hands over the control to the SIE at last.

For ping-pong buffer operation, the firmware can set one BD to send or receive the DATA0 packet, and the other for DATA1 packet without toggle (DTS = 0). That is, the even buffer is for DATA0 and the odd buffer is for DATA1.

3 USB Device Development

3.1 Hardware Design

Hardware design for USB device with MC9S08JM60 is simple and flexible. The designer can choose to use the on-chip regulator or external power supply and to use the internal pullup or external pullup resistor. You can use the internal pullup or external pullup resistor. The MCU provides many interfaces such as SPI, SCI, IIC, ADC, KBI, etc. The designers thus have more selections in design. The hardware design of the USB interface is discussed in the following sections.

3.1.1 Clocking Generation

USB module requires two clock sources. They are the 24 MHz bus clock and 48 MHz reference clock. According to the MCG features, the MCG must work in PEE mode (PLL engaged external) and an external clock source is essential for matching this requirement. An oscillator or crystal is used as the external clock source.

Example 1 shows the MCG initialization routine by using an external 12 MHz crystal. The system clock is switched from FEI mode to FBE mode and then to PBE mode and finally to PEE mode. The MCG status register (MCGSC) is checked in each step to ensure that the clock is stable, locked by PLL (or FLL), and switched successfully.

Example 1. MCG Initialization Routine Using External 12 MHz Crystal

```
void MCG_Init()
{
      /* the MCG is set to FEI mode by default, it should be change to FBE mode at first */
      MCGC2 = 0x36;          /*Select high frequency, high gain, Oscillator requested*/
      while(!(MCGSC & 0x02));                   /*Wait for the stable of Oscillator */
      MCGC1 = 0x9B;        /*External clock is selected, RDIV = 0b011 (to get 1.5MHz)*/
      while((MCGSC & 0x1C) != 0x08);
                             /*Check whether the external reference clock is selected */

      /* Switch to PBE mode from FBE*/
      MCGC3 = 0x48;                        /*Enable PLL, VDIV = 0b1000, multiply by 32*/
      while ((MCGSC & 0x48) != 0x48);                 /*Wait for the PLL to be locked */

      /*Switch to PEE mode from PBE mode*/
      MCGC1 &= 0x3F;                                          /*PLL output is selected*/
      while((MCGSC & 0x6C) != 0x6C);
                           /*Wait for the PLL output becoming the system clock source */
      return;
}
```

3.1.2 USB Device Power

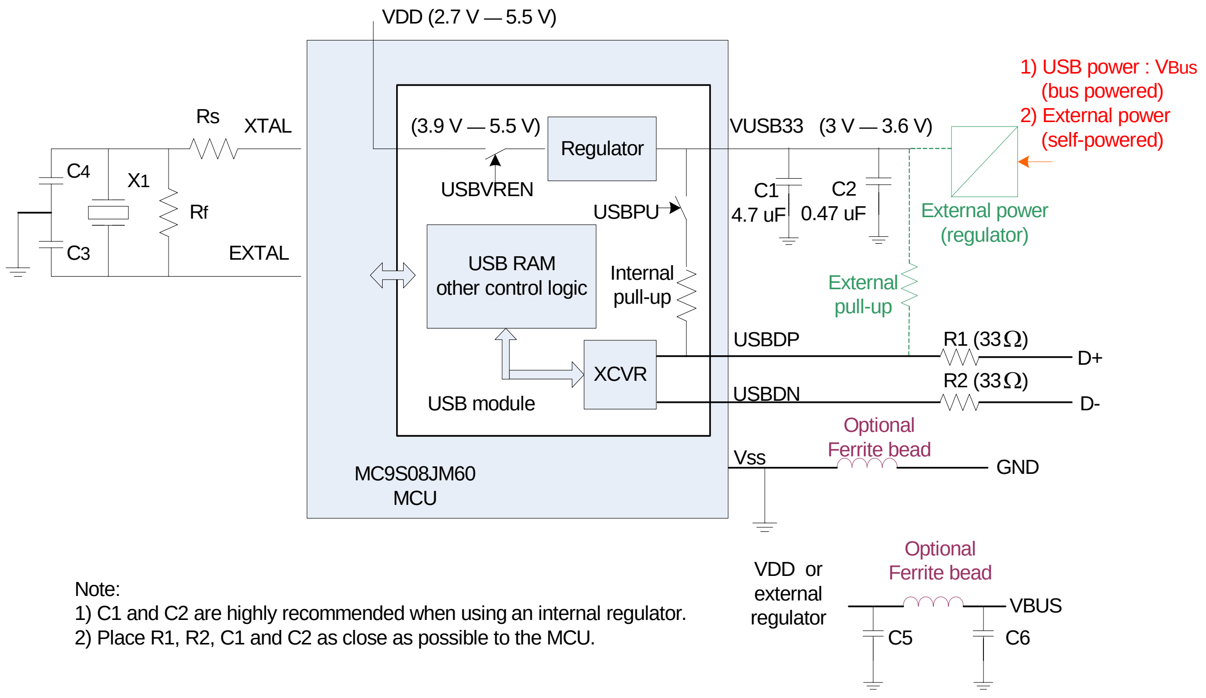

VDD (2.7 V — 5.5 V)
1) USB power : VBus (bus powered)
2) External power (self-powered)
Rs
XTAL
(3.9 V — 5.5 V)
Regulator
VUSB33 (3 V — 3.6 V)
C4
X1
Rf
C3
EXTAL
USBVREN
USBPU
C1 4.7 uF
C2 0.47 uF
External power (regulator)
USB RAM other control logic
Internal pull-up
External pull-up
USBDP
R1 (33Ω)
D+
XCVR
USB module
USBDN
R2 (33Ω)
D-
Optional Ferrite bead
Vss
GND
MC9S08JM60 MCU
VDD or external regulator
Optional Ferrite bead
VBUS
C5
C6
Note:
1) C1 and C2 are highly recommended when using an internal regulator.
2) Place R1, R2, C1 and C2 as close as possible to the MCU.

Figure 10. USB Connection Example

The dedicated on-chip regulator powers the USB transceiver. It shares the same power supply with MCU. The on-chip regulator is enabled or disabled by the USBVREN bit in the USBCTL0 register. When the on-chip regulator is disabled (USBVREN = 0), one external 3.3 V power is connected to $V_{USB3.3}$ pin to provide the power for USB transceiver and pullup resistor.

The nominal input range of this regulator is from 3.9 V to 5.5 V, but the MCU works at lower voltage (2.7 V – 5.5 V). When the power supply for MCU is below 3.9 V, an external power supply must be provided for the USB module and the internal regulator must be disabled.

In addition, it is highly recommended that two capacitors are connected to the $V_{USB\ 3.3}$ pin to decrease the ripple on the output of the regulator. Their recommended values are 0.47 μF, and 4.7 μF. (refer to Figure 10), and they must be placed as near as possible to the $V_{USB3.3}$ pin.

USB devices have two power modes — self-powered and the bus-powered. Refer to the USB specification for detailed information.

- Bus-Powered

  Bus-powered USB devices are powered from the USB bus. The maximum current drawn from the host is 500 mA. The USB device is powered when it is attached to USB bus and the connection is valid, then the host will find it attached.

- Self-Powered

  The self-powered USB device uses an individual power supply to provide the current for all components, so that its power consumption is limited by the capability of external power supply. The USB device whose current consumption is more than 500 mA must adopt self-powered mode.

  In self-powered applications, the MCU works without the USB connection. The firmware has the capability to know the status of USB connection (attached or not) for switching its state machine correctly. One GPIO pin or KBI pin is recommended to help detect the USB power; then the firmware determines the connection status.

  In addition, connect the ferrite bead in series between the power (or ground) of the host and USB device to help to absorb the high frequency noise coupled in the power system (decrease electromagnetic interference). The ferrite bead is similar to a high permeability inductance wherein the high frequency impedance attenuates the high-frequency current, and the DC current passes freely.

3.1.3 Pullup Resistor

USB host uses the pullup resistor to detect the connection of the USB device and identifies the device's speed (low, full, high). The USB module of MC9S08JM60 choose an internal or external pullup resistor.

The internal pullup resistor is enabled by the USBPU bit of the USBCTL0 register.

When the internal pullup resistor is disabled, one 1.5 kΩ pullup resistor can be connected between the $V_{USB3.3}$ pin and the USBDP pin. See the connection marked with dash line in Figure 8.

NOTE

The USB device can use internal pullup resistor or external pullup resistor, but only one pullup resistor is allowed to be connected.

Table 1 lists all the combinations USB regulator and pullup resistor. The designers can select one of the combinations according to actual requirement.

Table 1. USB Regulator and Pullup Resistor Configuration

USBPU	USBVREN	Pullup Resistor and Regulator
0	0	External pullup resistor, external 3.3 power supply
1	0	Internal pullup resistor, external 3.3 power supply
0	1	External pullup resistor, internal regulator
1	1	Internal pullup resistor, internal regulator

3.2 USB Firmware Design

3.2.1 USB Communication Model

On the high layer of the USB data flow model, the USB pipe is built for communication between the USB device and the host. A USB pipe is an association between an endpoint on a device and software on the host. Pipes represent the ability to move data between software on the host via a memory buffer and an endpoint on a device.

USB devices must build at least one control pipe to the host. The control pipe (also called default control pipe) is essential for all USB devices. It is based on endpoint 0, uses control transfer type, and supports bidirectional data communication. The other pipes can be designed in terms of the application's requirements. There are seven pipes available (seven endpoints) for MC9S08JM60 to support control, bulk, interrupt, or isochronous transfer.

The default control pipe is used by the USB system software to determine device identification and configuration requirements and to configure the device. The default control pipe can also be used by device-specific software after the device is configured. The USB enumeration and configuration is the first step to design the firmware of a USB device.

The USB communication consists of transfers. The USB transfer comprises one or more transactions, and each transaction includes several packets. For MC9S08JM60 USB devices, the firmware immediately needs to do little work. One transaction is completed by the USB module automatically. The programer can ignore work related to low level USB protocol, such as CRC calculation, building packet, and handshake. The programer can spend more effort on high-level protocol and data process.

The firmware design of USB device with MC9S08JM60 is discussed in the following sections. Detailed information about enumeration and configuration has also been covered.

3.2.2 Main Firmware Structure

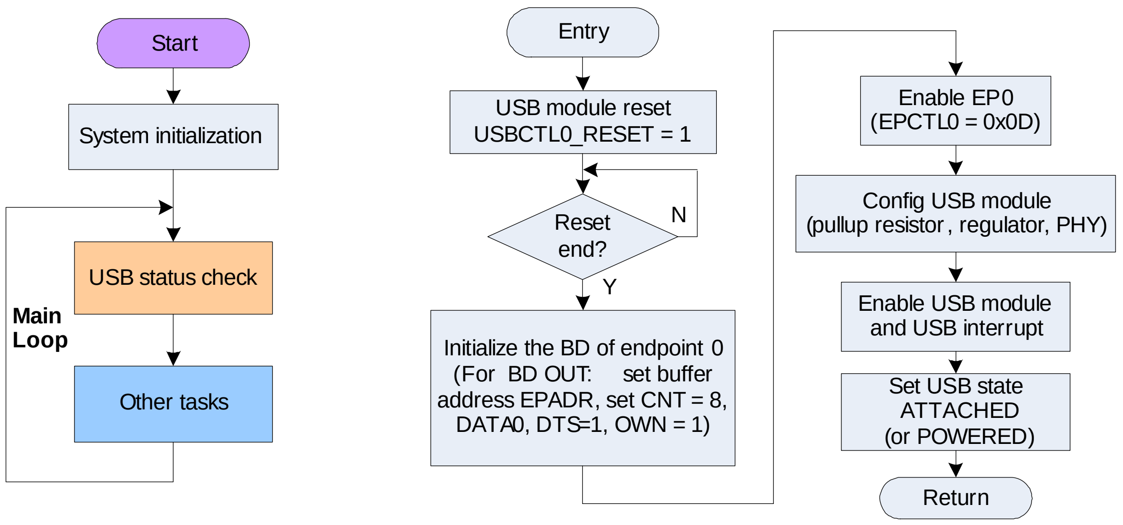

Start
System initialization
USB status check
Other tasks
Main Loop
Entry
USB module reset USBCTL0_RESET = 1
Reset end?
N
Y
Initialize the BD of endpoint 0 (For BD OUT: set buffer address EPADR, set CNT = 8, DATA0, DTS=1, OWN = 1)
Enable EP0 (EPCTL0 = 0x0D)
Config USB module (pullup resistor, regulator, PHY)
Enable USB module and USB interrupt
Set USB state ATTACHED (or POWERED)
Return

Figure 11. Example of the Main Function and USB Module Initialization

Figure 11 shows two flow charts of a USB firmware example designed with MC9S08JM60 device. One is the main routine, and the other is the USB module initialization.

After the MCU is powered on the firmware performs system initialization. All modules used in application are initialized one by one.

The firmware can complete the USB module initialization by following the steps in Figure 11. It initializes the USB BDT according to the USB RAM assignment, then initializes the USB registers, and sets the USB state to ATTACHED at last.

To set the USB registers, it configures the USB module (pullup resistor, USB regulator in terms of the hardware design) first, then enables the EP0 and the USB interrupt. For self-powered devices, the USB module can be enabled when MCU detects the USB bus power, and the USB device initial state is POWERED.

NOTE

The EPCTL0 must be set to 0x0D for control transfer on endpoint 0 before the enumeration starts.

In the main loop of firmware, the program checks the USB status changes, e.g., attached or detached to USB bus. The other customize tasks are also located in main loop.

The firmware can use the polling or interrupt method to deal with USB events. With the polling method, all USB status bits in the INTSTAT register are checked, the program will jump to associated service routine if the flag bits are set. The USB interrupt must be enabled before entering suspend mode if the MCU needs to work in stop3 mode.

If the polling method is not adopted, the interrupt method is a good choice. All events involved in the USB communication can be processed in real-time by the USB interrupt service routine, and all unnecessary polling for the status flag bits when no USB events occurred are avoided.

Figure 12 is an example of USB interrupt service routine (ISR). The USB interrupt flags in the INTSTAT register are checked one by one. When one USB event occurs, the associated interrupt flag is set, then the program will process these events in ISR if this interrupt is enabled. The program will check the next interrupt flag.

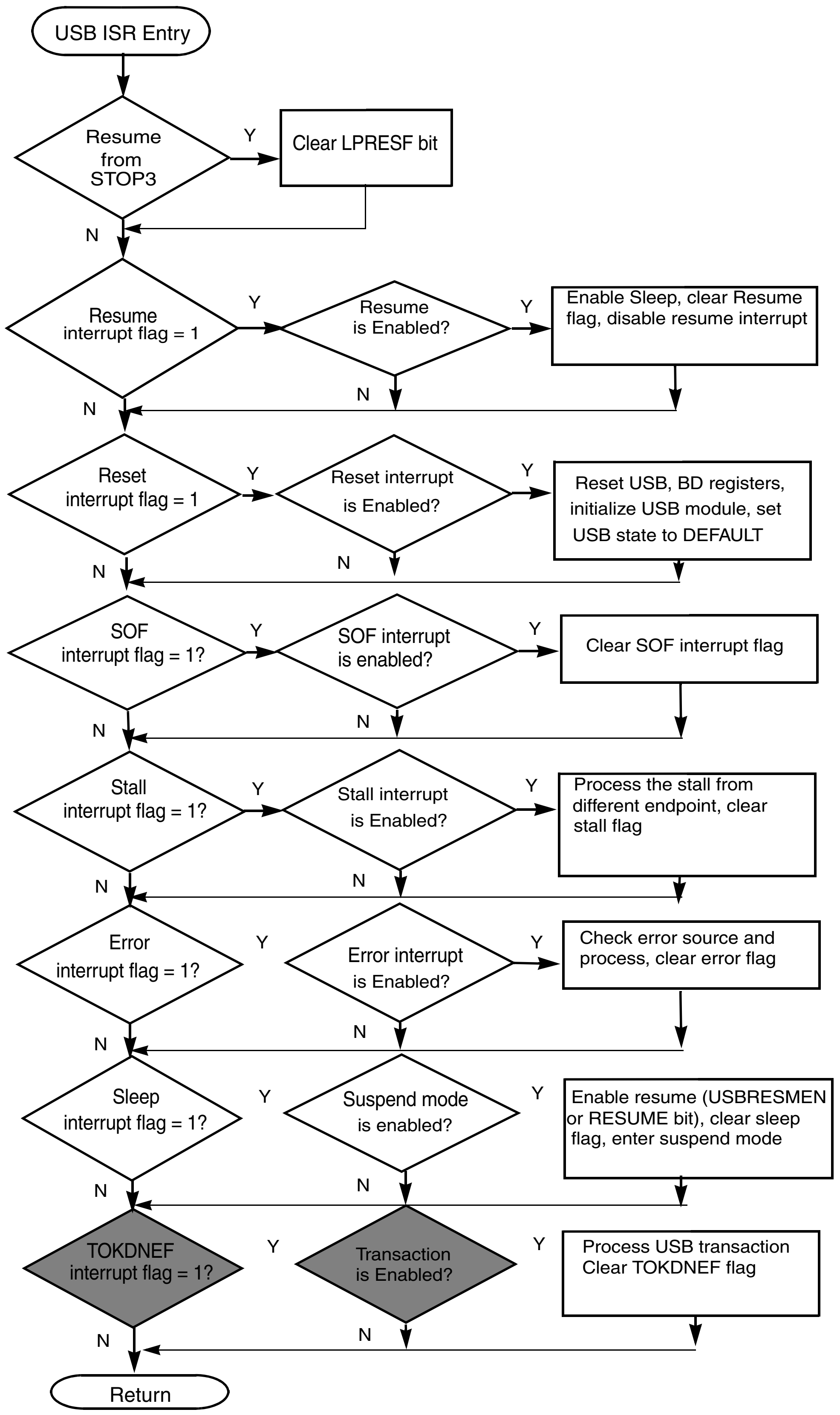

USB ISR Entry
Resume from STOP3
Y
Clear LPRESF bit
N
Resume interrupt flag = 1
Y
Resume is Enabled?
Y
Enable Sleep, clear Resume flag, disable resume interrupt
N
N
Reset interrupt flag = 1
Y
Reset interrupt is Enabled?
Y
Reset USB, BD registers, initialize USB module, set USB state to DEFAULT
N
N
SOF interrupt flag = 1?
Y
SOF interrupt is enabled?
Y
Clear SOF interrupt flag
N
N
Stall interrupt flag = 1?
Y
Stall interrupt is Enabled?
Y
Process the stall from different endpoint, clear stall flag
N
N
Error interrupt flag = 1?
Y
Error interrupt is Enabled?
Y
Check error source and process, clear error flag
N
N
Sleep interrupt flag = 1?
Y
Suspend mode is enabled?
Y
Enable resume (USBRESMEN or RESUME bit), clear sleep flag, enter suspend mode
N
N
TOKDNEF interrupt flag = 1?
Y
Transaction is Enabled?
Y
Process USB transaction Clear TOKDNEF flag
N
N
Return

Figure 12. Example of USB Interrupt Service Routine

When the USB device is in suspend mode, the resume interrupt flag and the resume interrupt processing differ according to the work mode of MCU (stop3 or user mode). The LPRESF bit in the USBCTL0 register is set for resuming from stop3 mode, and the RESUMEF bit in the INTSTAT register is set for resuming from the user mode. The processing is listed in Figure 12. The designer can select from these two ways. This is the reason why the processing for RESUMEF is marked with dotted line.

The RESET bit is set when the host sends a reset signal to the device. The program for processing a RESET must stop all USB involved processes immediately, reset the USB registers (e.g, ADDR=0), and return to the state before enumeration begins.

The TOKDNEF bit is set when one transaction is complete. The processing of TOKDNEF is the main entry for all transactions.

The interrupt flag must be cleared after it is serviced. Otherwise the interrupt is executed again and again, without stopping.

3.2.3 USB State Machine

USB device has the following states:

- Powered: USB device is not attached to the USB bus, but the MCU has been powered on.
- Attached: USB device is attached to the USB bus, but enumeration has not started
- Default: USB device is reset by the host
- ADDR_PENDING: USB device receives the Set_Address command from the host
- Addressed: The Set_Address command has executed successfully.
- Configured: The USB device receives and executes the Set_Configuration command successfully.
- Suspend: The USB device enters suspend mode.

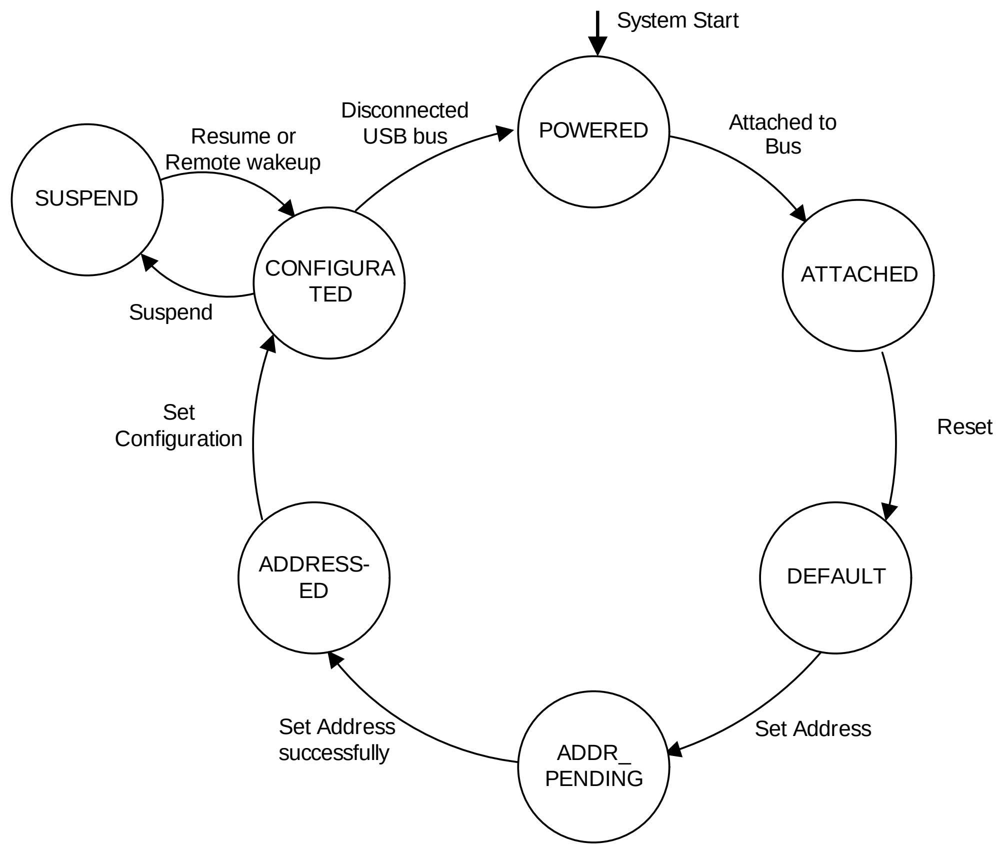


Figure 13. USB Device State Transfer

Figure 13 shows the switches of the USB state machine. The suspend and reset modes can be entered from all states except the powered state.

The bus-powered USB device sets its state to attached directly after it is attached to the USB bus.

The state of ADDR_PENDING is set when USB device receives the Set_Address setup token. It is changed to ADDRESSED when the Set_Address transfer has finished.

After the USB device is attached to the USB bus, the host starts the enumeration process, the USB device state changes to configured from attached if the enumeration process succeeds.

3.2.4 USB Device Enumeration Process

In the enumeration process, the host performs the identification and configuration for the USB device. The following process is an example for enumeration between a USB device and computer with Windows XP operating system installed.

1. The host detects the connection of a new device via the device’s pullup resistors on the data pair. The host waits for at least 100 ms, allowing for the plug to be inserted fully and for power to be stable on the device.
2. Host issues a reset to make the device in default state.

3. The MS Windows host asks for the first 64 bytes of the device descriptor.
4. After receiving the first eight bytes of the device descriptor, host immediately issues another bus reset.
5. The host now issues a Set_Address command to place the device in the addressed state.
6. The host asks for the entire 18 bytes of the device descriptor.
7. The host then asks for nine bytes of the configuration descriptor to determine the overall size.
8. The host asks for 256 bytes of the configuration descriptor.
9. Host asks for any string descriptors specified.
10. Windows will ask for a driver for device. It may request all the descriptors again before it issues a Set configuration request.
11. Host sets the configuration of device. The USB device state changes to configured.

The enumeration process is similar for all USB devices. The host fetches a lot of descriptors from the device, such as the devices descriptor, configuration descriptor, interfaces descriptors, endpoint descriptors, and string descriptors. According to the information in these descriptors, the operating system finds the driver for the USB device, and the driver sets the configuration of device at last.

The enumeration process is carried on the endpoint 0, which the default control pipe is built in. The control transfers constitute the enumeration process, so the program about control transfer is the most important part for USB stack.

The implementation for control transfer on endpoint 0 has been described in the sections that follow. The setting for the BD registers and the endpoint buffer is also involved.

3.2.4.1 USB RAM Allocation and Access

BDT is located at the beginning of the USB RAM. Because the location of BD assigned for different endpoint is fixed, you can define the following data structure for BDT access (defined with C language).

Example 2. Data Structure for BDT Access

```
typedef union _BD_STAT
{
      byte _byte;
      struct{
          unsigned :1;
          unsigned :1;
          unsigned BDTSTALL:1;                            /*Buffer Stall Enable*/
          unsigned DTS:1;                            /*Data Toggle Synch Enable*/
          unsigned :1;                              /*Address Increment Disable*/
          unsigned :1;                                          /*BD Keep Enable*/
          unsigned DATA:1;                            /*Data Toggle Synch Value*/
          unsigned OWN:1;                                        /*USB Ownership*/
      }McuCtlBit;
      struct{
          unsigned :1;
          unsigned :1;
          unsigned PID0:1;
          unsigned PID1:1;
          unsigned PID2:1;
```

```
            unsigned PID3:1;
            unsigned :1;
            unsigned OWN:1;
        }SieCtlBit;
        struct{
            unsigned    :2;
            unsigned PID:4;                                        /*Packet Identifier*/
            unsigned    :2;
        }RecPid;
} BD_STAT;       /*Buffer Descriptor Status Register*/
typedef struct _BUFF_DSC
{
        BD_STAT Stat;
        byte Cnt;
        byte Addr;                                                   /*Buffer Address*/
} BUFF_DSC;                                                /*Buffer Descriptor Table*/

typedef struct _BDTMAP
{
        BUFF_DSC ep0Bi;                                             /*Endpoint 0 BD In*/
        BUFF_DSC ep0Bo;                                            /*Endpoint 1 BD Out*/
        BUFF_DSC ep1Bio;                                     /*Endpoint 1 BD IN or OUT*/
        BUFF_DSC ep2Bio;                                     /*Endpoint 2 BD IN or OUT*/
        BUFF_DSC ep3Bio;                                     /*Endpoint 3 BD IN or OUT*/
        BUFF_DSC ep4Bio;                                     /*Endpoint 4 BD IN or OUT*/
        BUFF_DSC ep5Bio_Even;                          /*Endpoint 5 BD IN or OUT  Even*/
        BUFF_DSC ep5Bio_Odd;                            /*Endpoint 5 BD IN or OUT  Odd*/
        BUFF_DSC ep6Bio_Even;                          /*Endpoint 6 BD IN or OUT  Even*/
        BUFF_DSC ep6Bio_Odd;                            /*Endpoint 6 BD IN or OUT  Odd*/
        WORD Reserved;
}BDTMAP;
BDTMAP Jm60_Bdt @0x1860;
byte  EP0_In_Buf[8] @0x1880;
byte EP0_Out_Buf[8] @0x1888;
byte EP1_In_Buf[64] @0x1890;
```

In this example, all bits in the BD registers are declared in the BD_STAT structure. The BDTMAP structure includes all BDs for seven endpoints. The structure variable Jm60_Bdt is declared, and its address is fixed at 0x1860. According to the memory allocation rules for structure, all the BD registers for different endpoint points to their location in USB RAM.

Firmware accesses the OWN bit of the BD registers like this,

```
Jm60_Bdt.ep1Bio.Stat.McuCtlBit.OWN  = 1; /* hand over the control to SIE*/.
```

Firmware defines the endpoint buffers and assigns them in USB RAM, that can be accessed directly.

EP0_Out_Buf[8] is defined and fixed to 0x1888.

3.2.4.2 Control Transfer

Control transfer has three stages. They are the setup, data, and status stages. To control transfer, the data stage can be omitted, e.g, Set_Address.

The data transferring direction in data stage is determined by the request type transferred to the USB device in the setup stage. The direction in the status stage is different from that in the data stage. If the direction in data stage is in (out), the direction in the status stage is out (in).

The data packet type for control transfer begins with DATA0, and toggles continuously in the setup and data stages. The data packet type must be DATA1 in the status stage even if the last data type in data stage is DATA1.

According to the data transfer direction in different stages, three states describe the control transfer.

- WAIT_SETUP_TOKEN: before the beginning of the control transfer
- CTL_TRF_DATA_TX: send data to the host (in direction) in current or the next transaction
- CTL_TRF_DATA_RX: receive data from the host (out direction) in current or the next transaction

The control transfer begins with a SETUP token issued by the host. Before the SETUP token is received by device, the state is WAIT_SETUP_TOKEN. The firmware determines the next state is CTL_TRF_DATA_TX or CTL_TRF_DATA_RX according to the request type in the first transaction. The state transfer is demonstrated in Figure 14.

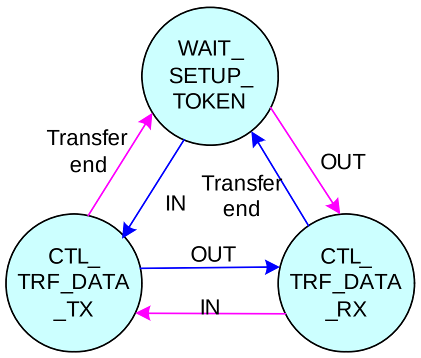
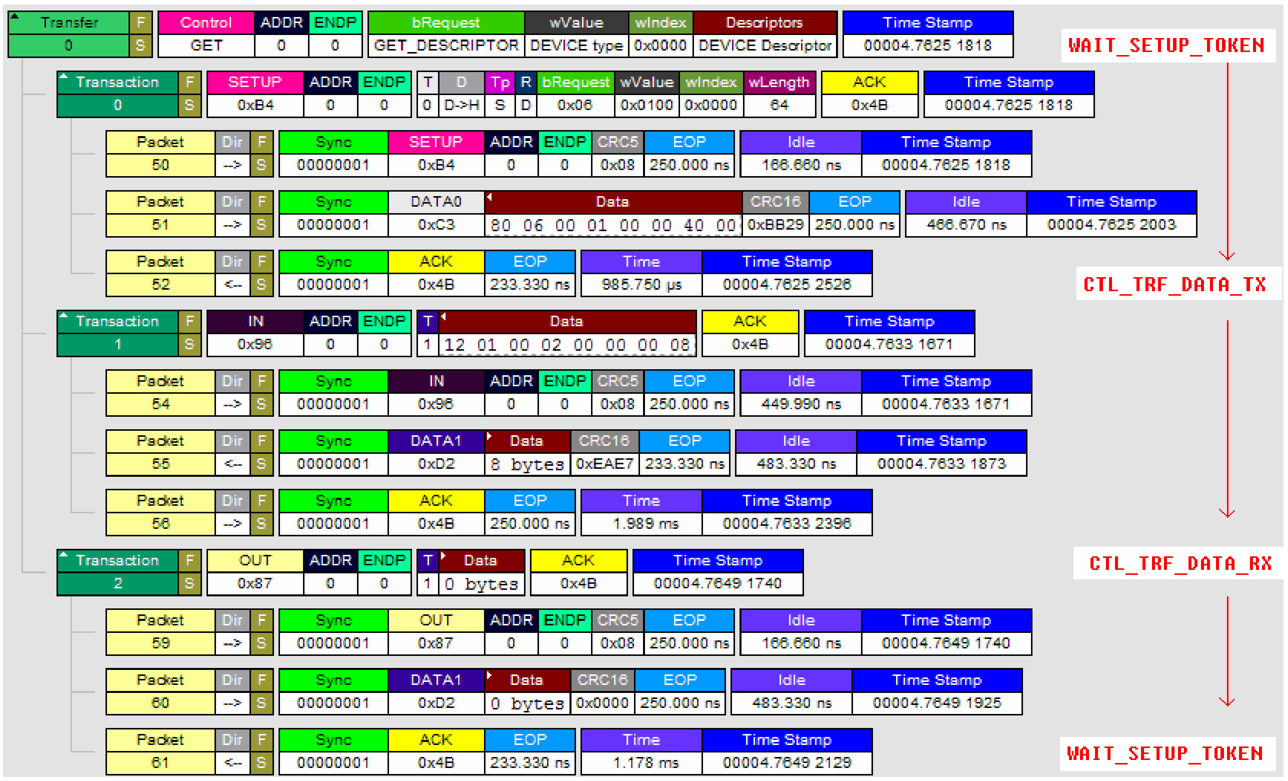


Figure 14. State Transfer for Data Transfer Direction

Figure 15 is a USB control transfer example. The transfer in this chart is issued by the host to get the device descriptor. It comprises three transactions. The first transaction belongs to the setup stage. The second and the third respectively attribute to the data stage and the status stage.



Transfer	F	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time Stamp
0	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	00004.7625 1818

Transaction	F	SETUP	ADDR	ENDP	T	D	Tp	R	bRequest	wValue	wIndex	wLength	ACK	Time Stamp
0	S	0xB4	0	0	0	D->H	S	D	0x06	0x0100	0x0000	64	0x4B	00004.7625 1818

Packet	Dir	F	Sync	SETUP	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
50	-->	S	00000001	0xB4	0	0	0x08	250.000 ns	166.660 ns	00004.7625 1818

Packet	Dir	F	Sync	DATA0	Data	CRC16	EOP	Idle	Time Stamp
51	-->	S	00000001	0xC3	80 06 00 01 00 00 40 00	0xBB29	250.000 ns	466.670 ns	00004.7625 2003

Packet	Dir	F	Sync	ACK	EOP	Time	Time Stamp
52	<--	S	00000001	0x4B	233.330 ns	985.750 µs	00004.7625 2526

Transaction	F	IN	ADDR	ENDP	T	Data	ACK	Time Stamp
1	S	0x96	0	0	1	12 01 00 02 00 00 00 08	0x4B	00004.7633 1671

Packet	Dir	F	Sync	IN	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
54	-->	S	00000001	0x96	0	0	0x08	250.000 ns	449.990 ns	00004.7633 1671

Packet	Dir	F	Sync	DATA1	Data	CRC16	EOP	Idle	Time Stamp
55	<--	S	00000001	0xD2	8 bytes	0xEAE7	233.330 ns	483.330 ns	00004.7633 1873

Packet	Dir	F	Sync	ACK	EOP	Time	Time Stamp
56	-->	S	00000001	0x4B	250.000 ns	1.989 ms	00004.7633 2396

Transaction	F	OUT	ADDR	ENDP	T	Data	ACK	Time Stamp
2	S	0x87	0	0	1	0 bytes	0x4B	00004.7649 1740

Packet	Dir	F	Sync	OUT	ADDR	ENDP	CRC5	EOP	Idle	Time Stamp
59	-->	S	00000001	0x87	0	0	0x08	250.000 ns	166.660 ns	00004.7649 1740

Packet	Dir	F	Sync	DATA1	Data	CRC16	EOP	Idle	Time Stamp
60	-->	S	00000001	0xD2	0 bytes	0x0000	250.000 ns	483.330 ns	00004.7649 1925

Packet	Dir	F	Sync	ACK	EOP	Time	Time Stamp
61	<--	S	00000001	0x4B	233.330 ns	1.178 ms	00004.7649 2129

Figure 15. Example Of Control Transfer and its State Changes

The state changes in control transfer are marked on the right of Figure 15.

The DATA toggle (DATA0/1) for control transfer can also be found. Its changing sequence is DATA0 (first transaction, the setup stage) DATA1 (second transaction, the data stage) DATA1 (third transaction, the status stage).

The SOF packets are hidden in Figure 15, so some packets like the 53, 57 and 58 packets cannot be found.

From the image displayed above, the firmware can determine the change from the data stage to the status stage by detecting the switch of the transaction direction (in/out). The in and out direction are controlled by the host, the change means the data stage is complete, and the control transfer state is also changed.

The firmware can also trace the state change between CTL_TRF_DATA_TX and CTL_TRF_DATA_RX by calculating the data length transferred or received. If all bytes have been transferred or received, the data stage is complete.

However this is not a good way to determine that the data stage is complete. The USB device cannot make sure that the host has received all those data packets. It is not recommended to use this method to change the control transfer state.

3.2.4.3 Process of USB Control Transfer

USB control transfer comprises two or more transactions. If an endpoint is correctly set, the transactions can be automatically controlled and processed by SIE. The TOKDNEF bit is set when one transaction completes. Figure 16 shows the process of one complete transaction carried on endpoint 0, it belongs to the enumeration process.

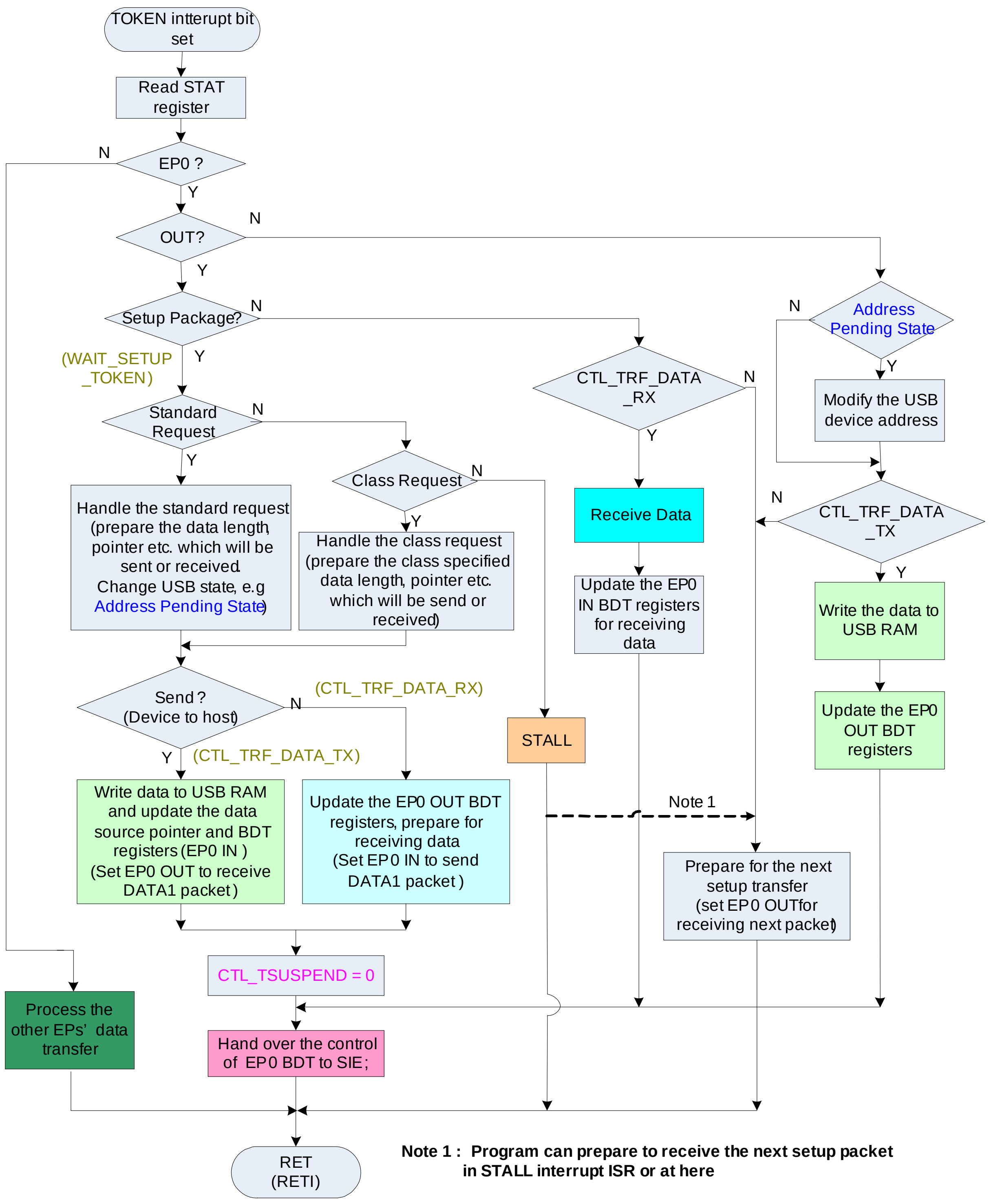

TOKEN intterupt bit set
Read STAT register
EP0 ?
N
Y
OUT?
N
Y
Setup Package?
N
(WAIT_SETUP _TOKEN)
Y
Standard Request
N
Y
Class Request
N
Y
Handle the standard request (prepare the data length, pointer etc. which will be sent or received. Change USB state, e.g Address Pending State)
Handle the class request (prepare the class specified data length, pointer etc. which will be send or received)
Send ? (Device to host)
N
(CTL_TRF_DATA_RX)
Y
(CTL_TRF_DATA_TX)
Write data to USB RAM and update the data source pointer and BDT registers (EP0 IN) (Set EP0 OUT to receive DATA1 packet)
Update the EP0 OUT BDT registers, prepare for receiving data (Set EP0 IN to send DATA1 packet)
CTL_TSUSPEND = 0
Hand over the control of EP0 BDT to SIE;
Process the other EPs' data transfer
RET (RETI)
STALL
CTL_TRF_DATA _RX
N
Y
Receive Data
Update the EP0 IN BDT registers for receiving data
Note 1
Prepare for the next setup transfer (set EP0 OUTfor receiving next packet)
Address Pending State
N
Y
Modify the USB device address
CTL_TRF_DATA _TX
N
Y
Write the data to USB RAM
Update the EP0 OUT BDT registers
Note 1 : Program can prepare to receive the next setup packet in STALL interrupt ISR or at here

Figure 16. Transaction of TOKDNEF

The TOKDNEF bit in the USB interrupt flag register is set first. The firmware reads the STAT register to determine whether the transaction occurs on endpoint 0 or not. If it is not on endpoint 0, the firmware will jump to the associated routine for that endpoint. If it is on endpoint 0, the firmware will check the transaction's direction (in or out).

If the transaction is in direction, the program will verify whether the received package is a SETUP package or not. If yes, the program determines the data source (or object) pointer and length to be transferred (or received) according to the request type. The request type (USB standard request or class request) determines the data transfer direction and the address and length of source or object data. After that, the control transfer state is adjusted to CTL_TRF_DATA_TX (or CTL_TRF_DATA_RX) from WAIT_SETUP_TOKEN by the data transfer direction bit in the USB request.

If the request is not supported, the firmware will stall the current transfer by setting the STALL bit in the EPCTL register (EPSTALL) or the BD control and status register (BDTSTALL). The USB module will then prepare the next control transfer in the STALL interrupt service routine or at the location marked with dotted line.

After that, if the direction is in, the firmware will write the data to the buffer according to the maximum length supported by the endpoint. The BD registers are also updated according to the requirements.

At last, the CTL_TSUSPEND bit in the CTL register is cleared for processing the next packet and the MCU hands over the control to SIE. This routine ends (RETI or RET).

If the transaction is out direction, and it is not for SETUP, the firmware will check if the control transfer is CTL_TRF_DATA_RX. If the result is true, the firmware will read out the data from the endpoint out buffer according to the length in the BC register. Then firmware will prepare for the next transaction.

If the control transfer state is CTL_TRF_DATA_TX and this transaction is in the end of the status stage of current transfer, this transfer has finished. The firmware needs to begin the preparation for the next transfer.

If this transaction is for endpoint 0, we add a supplementary check for the USB state. When the USB Set_Address standard request is issued, the USB state is changed to ADDR_PENDING in the first transaction of this transfer. An in transaction is issued now.

In the status stage of Set_Address transfer, the new USB device address is written to the USB address register. If the USB state is not ADDR_PENDING, the firmware will check the current control transfer state. If it is true, the firmware will write the data to be delivered to the endpoint buffer, then update the BD registers for transferring. If it is false, this transaction belongs to the status stage, current transfer ends.

The interrupt flag must be cleared in the service routine by writing 1 to TOKDNEF bit.

The program illustrated Figure 16 can be implemented with the interrupt method or the polling method. In both methods, each interrupt flag bit in the INTSTAT register is checked and processed.

3.2.5 USB Transactions on EP1-6

Firmware designer can assign the endpoint direction, buffer size, location, and transfer type according to the requirements of the application. Some information is included in the endpoint descriptor and reported to the host in the enumeration process.

When a transaction is complete in any one of the endpoint 1– 6, the TOKDNEF bit is set, and the firmware can process it like the transactions in control transfer. In Figure 16, if the firmware finds the transaction did not occur on endpoint 0, it jumps to an associated routine for a different endpoint.

Example 3. Processing for Transaction on Different Endpoints

```
void USB_Transaction_Handler(void)
{
      unsigned char stat = STAT;

      if((stat & 0xF0)  == 0x00)
      {
      ......                      /*the processing for the transaction on endpoint 0*/
      }
      else
      {
          if((stat & 0xF0) == 0x20)
              HID_Rec_Data();
          /*add the code for other endpoints*/

      }
      return;
}
```

In Example 3, the firmware processes the transaction on a different endpoint by checking the STAT register. The code for other endpoints can be inserted like the HID_Rec_Data routine. The HID_Rec_Data is used to process the received data from the host on endpoint 2.

All endpoints and their operations have similar BD registers and endpoint buffers. The control, interrupt, and bulk transfer operation on these endpoints are also similar. The isochronous transfer is different. It does not need to set the EPHSHK bit in the EPCTLx register and toggle the DATA0/1, because low-level USB protocol does not allow handshakes to be returned to the transmitter of an isochronous pipe. For isochronous transfers, the protocol is optimized by assuming transfer normally succeeds because timeliness is more important than correctness/retransmission.

3.2.6 Suspend and Resume

USB host can make the USB device or whole USB bus enter the suspend state at any time. As per the USB specification, the USB device enters suspend mode when the USB bus is idle for more than 3 ms. This process must be completed by the MCU in 7 ms, after the first 3 ms have elapsed.

The current consumed by the USB device from the USB bus must be less than 500 μA for a low-power device (2.5 mA for a high-power device). The MC9S08JM60 MCU supports low-power devices. To achieve the current consumption in suspend mode, the MC9S08JM60 MCU must work in stop3 mode if it is bus-powered. It is not essential if the USB device is self-powered.

For the bus-powered USB device, the USBRESMEN bit in the USBCTL0 register must be set to 1 before the MCU enters stop3 mode. This enables an asynchronous interrupt to wake up the MCU from stop3 mode. For the self-powered device, firmware can set REUSME in the INTENB register to enable USB resume if the MCU does not need to enter stop3 mode.

There are two kinds of resume methods for the MC9S08JM60 MCU resumed by the host and remote-wakeup.

The host sends out the resume signal (K state for at least 20 ms), then the RESUMEF flag in the INTSTAT register is set. The USB interrupt is triggered when the MCU is in work mode and the RESUME is enabled.

The LPRESF bit in the USBCTL0 register is set when USBRESME bit is set and MCU is in stop3 mode. At the same time, an asynchronous interrupt is triggered and brings the MCU out of stop3 mode.

This interrupt has the same vector as USB interrupt. The USBRESME bit must be cleared immediately in the interrupt service routine. In addition, the firmware must check whether the resume signal on USB bus is caused by noise. MCU must enter stop3 mode again if the noise causes this.

The USB module of the MC9S08JM60 MCU supports remote-wakeup. The remote-wakeup signal can be generated by setting the CRESUME bit in the CTL register. The firmware can set this bit, then clear it after a delay between 5 to 15 ms. This signal brings the USB bus out of suspend mode.

The following program (next page) is an example of USB suspend and resume (from stop3). The USB_Suspend routine is called when the suspend is issued by the host. The USBRESUMEN bit in the USBCTL0 register is set to enable the resume in stop3 mode, then the USB state is changed to USB_SUSPEND. USB_Suspend routine calls the USB_Wakeup to enter stop3 mode.

In stop3 mode, the MCU power consumption decreases greatly, to approximately 270 µA. This value is tested with 5 V power supply, KBI module enabled, USB in suspend mode, and all other modules are off.

In the USB_Wakeup routine, the USB backs to CONFIGURED state when the MCU is awakened by USB bus or other interrupt source. If the MCU is awakened by KBI interrupt, the USB_Wakeup routine returns 2 if it supports remote-wakeup. Otherwise it returns 0. The USB_Wakeup routine returns 1 if it is woken by USB bus resume signal. The delay and check for the RESUMEF bit helps to avoid the pseudo resume signal introduced by noise, because the RESUMEF has enough time to be set after the MCU is awakened and the clock is recovered.

The USB_Suspend routine determines to call the Usb_Remote_Wakeup routine, enters stop3 mode, or continues execution according to the return value of USB_Wakeup.

The other interrupt source can also bring MC9S08JM60 MCU out from stop3 mode. The firmware can use remote-wakeup to resume the USB bus or return to stop3 mode according to the application requirement. The RTC interrupt is disabled in Example 4 in case the MCU is waken up by RTC interrupt.

Example 4. USB Suspend and Resume Routine

```
void USB_Suspend(void)
{
      unsigned char Result;

      /* INTENB_RESUME = 1;   */                         /*MCU run mode, self-powered*/
      USBCTL0_USBRESMEN = 1;                                /*stop3 mode, Bus-powered*/
      Usb_Device_State = USB_SUSPEND;                             /*USB state changes*/
      /*return at here, if it is self-powered, and does not eneter into stop3*/
      RTC_DISBALE();                                                    /*disable RTC*/

      do
      {
          Result = USB_WakeUp() ;
      }while(!Result);


      if(Result == 2)
          USB_Remote_Wakeup();

      RTC_ENABLE();
      return;
}


unsigned char USB_WakeUp(void)
 {
      int delay_count;

      asm STOP;    /*MCU enters stop3*/

      Usb_Device_State = CONFIGURED_STATE;

      if((! USBCTL0_LPRESF) && (Kbi_Stat))
          {
          if(Usb_Stat.BitCtl.RemoteWakeup == 1)
              return 2;
          else
              return 0;
          }
      else
      {
          delay_count = 50;
          do
          {
              delay_count--;
          }while(delay_count);
          if(INTSTAT_RESUMEF)
          {
              return 1;
          }
          else
              return 0;
      }
}
```

If the USB device does not need to enter stop3 mode, the firmware can set RESUME bit, clear SLEEPF bit, and change the USB device state to suspend. After this, it can return to the main routine.

The LPRESF bit in USBCTL0 is checked in the USB interrupt service routine (USB_ISR routine in Example 5). It is set when the USB device is resumed from stop3 mode. The USBRESUMEN bit must be cleared immediately to clear LPRESF — low power resume bit in USB ISR. The SLEEPF bit is set when the USB bus is idle for more than 3 ms. The firmware sets the Usb_Device_State to WAIT_SUSPEND and clears the SLEEPF bit in the USB interrupt routine.

The firmware calls USB_Suspend routine to avoid entering stop in the USB interrupt service routine when the USB state changes to WAIT_SUSPEND. The WAIT_SUSPEND state is a middle state that is switched to suspend instantaneously, so it does not appear in the USB state machine.

The USB_Remote_Wakeup is called when the remote-wakeup signal is issued by the USB device. It sets the CRESUME bit in the CTL register to generate the remote-wakeup signal on the USB bus, then clears this bit in 15 ms.

Example 5. USB Suspend and Resume Processing and Remote Wakeup Routine

```
void interrupt 7 USB_ISR()
{
      if(USBCTL0_LPRESF)
      {
          USBCTL0_USBRESMEN = 0;/*clear this bit immediately
      }
      …… /*The processing for other USB interrupt flag*/
      if(INTSTAT_SLEEPF)
      {
          Usb_Device_State = WAIT_SUSPEND;
          INTSTAT_SLEEPF = 1;
      }
      return;
}
void USB_Remote_Wakeup(void)
{
      static word delay_count;

      USB_WakeFrom_Suspend();

      CTL_CRESUME = 1;                                         /* Start RESUME signal*/

      delay_count = 8000;                              /* Set RESUME line for 1-15 ms*/
      do
      {
          delay_count--;
      }when(delay_count);

      CTL_CRESUME = 0;                                         /* Clear RESUME signal*/
      return;
}
```

4 Summary

USB PHY implements the physical layer of USB protocol.

The SIE of the USB module implements the transferring and receiving logic. It can complete most of the USB low-level protocol automatically. The programer immediately needs to set some registers and read or write data.

All 256 bytes of USB RAM are separate from MCU RAM. It provides the space for USB BDT and data buffer of the endpoint. The ping-pong buffer can be used to improve the data throughput.

The USB device designer can use the on-chip or external regulator with the pullup resistor.

Based on all the features described in this document, the MC9S08JM60 USB module provides an easy way for the USB device development. The designer will spend considerably less time on the low level of USB protocol. This enables efficient design of the USB stack firmware.

The MC9S08JM60 USB firmware design has also been discussed. The illustration for firmware structure, USB initialization, USB interrupt processing, the USB device state machine, the implementation of control transfer for enumeration, and the USB suspend and resume provides a lot of information for USB stack design. With the HID example attached and detailed information that must be noted during the design process given in this document, the designers can port the USB stack to their applications and focus on the application layer.

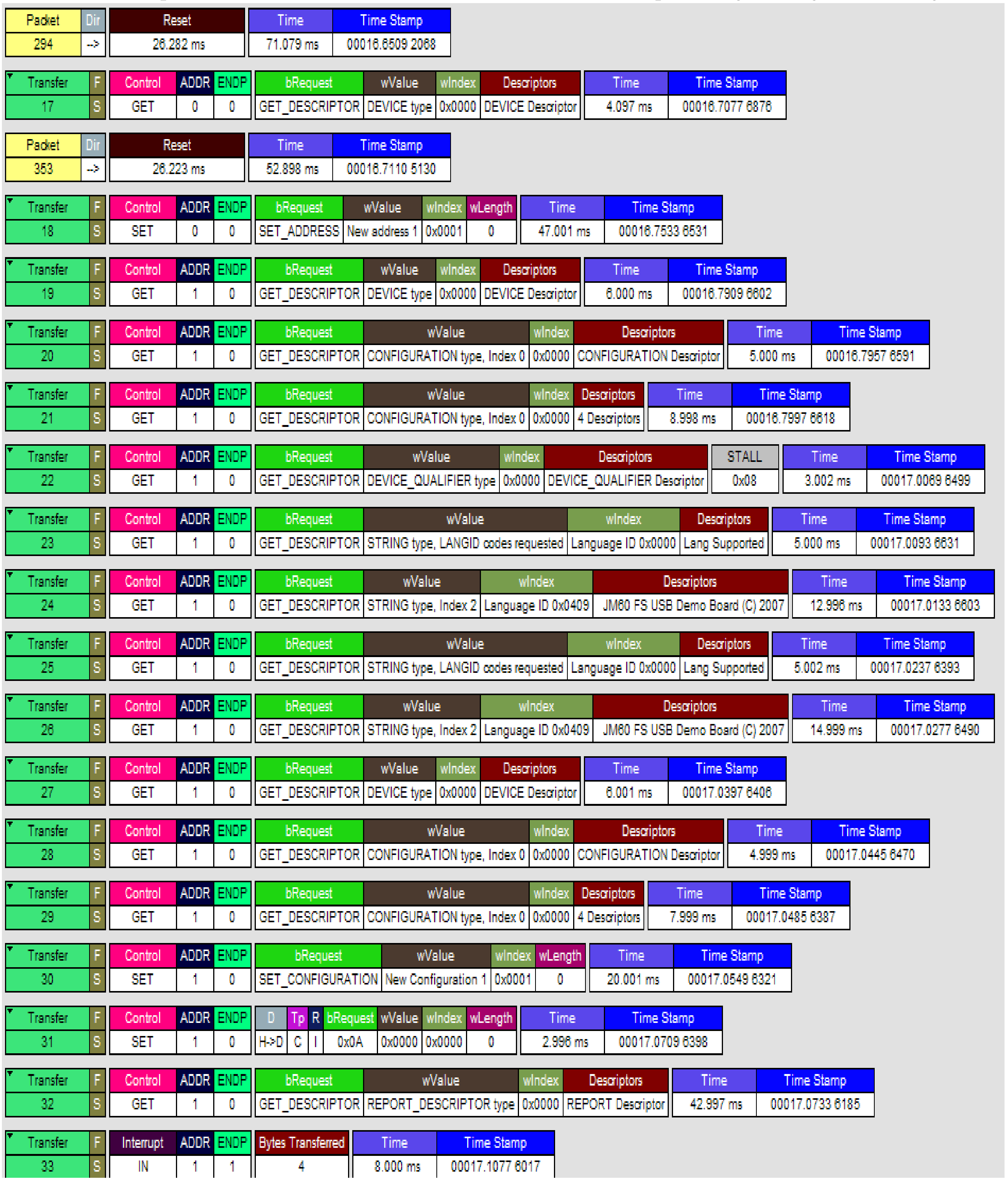
Appendix A Firmware for HID Mouse

An example project is provided for reference. The project is based on the MC9S08JM60 demo board and CodeWarrior for HC(S)08 V5.1 with the MC9S08JM60 service pack.

This project demonstrates the firmware design for an HID class device and most of the detailed information is illustrated in this application note. You can compile the project and download it into the MC9S08JM60 demo board. Attach it on the USB bus. The windows system (XP) can identify this device. After that, the demo board can work as a mouse.

Appendix B Enumeration Process of an HID Mouse

The enumeration process of HID mouse is shows as follows. It is captured by Lecroy USB Analyzer.



Packet	Dir	Reset	Time	Time Stamp
294	-->	26.282 ms	71.079 ms	00016.6509 2068

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
17	S	GET	0	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	4.097 ms	00016.7077 6876

Packet	Dir	Reset	Time	Time Stamp
353	-->	26.223 ms	52.898 ms	00016.7110 5130

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength	Time	Time Stamp
18	S	SET	0	0	SET_ADDRESS	New address 1	0x0001	0	47.001 ms	00016.7533 6531

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
19	S	GET	1	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	6.000 ms	00016.7909 6602

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
20	S	GET	1	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor	5.000 ms	00016.7957 6591

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
21	S	GET	1	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	4 Descriptors	8.998 ms	00016.7997 6618

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	STALL	Time	Time Stamp
22	S	GET	1	0	GET_DESCRIPTOR	DEVICE_QUALIFIER type	0x0000	DEVICE_QUALIFIER Descriptor	0x08	3.002 ms	00017.0069 6499

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
23	S	GET	1	0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	Lang Supported	5.000 ms	00017.0093 6631

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
24	S	GET	1	0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	JM60 FS USB Demo Board (C) 2007	12.996 ms	00017.0133 6603

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
25	S	GET	1	0	GET_DESCRIPTOR	STRING type, LANGID codes requested	Language ID 0x0000	Lang Supported	5.002 ms	00017.0237 6393

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
26	S	GET	1	0	GET_DESCRIPTOR	STRING type, Index 2	Language ID 0x0409	JM60 FS USB Demo Board (C) 2007	14.999 ms	00017.0277 6490

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
27	S	GET	1	0	GET_DESCRIPTOR	DEVICE type	0x0000	DEVICE Descriptor	6.001 ms	00017.0397 6406

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
28	S	GET	1	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	CONFIGURATION Descriptor	4.999 ms	00017.0445 6470

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
29	S	GET	1	0	GET_DESCRIPTOR	CONFIGURATION type, Index 0	0x0000	4 Descriptors	7.999 ms	00017.0485 6387

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	wLength	Time	Time Stamp
30	S	SET	1	0	SET_CONFIGURATION	New Configuration 1	0x0001	0	20.001 ms	00017.0549 6321

Transfer	F S	Control	ADDR	ENDP	D	Tp	R	bRequest	wValue	wIndex	wLength	Time	Time Stamp
31	S	SET	1	0	H->D	C	I	0x0A	0x0000	0x0000	0	2.996 ms	00017.0709 6398

Transfer	F S	Control	ADDR	ENDP	bRequest	wValue	wIndex	Descriptors	Time	Time Stamp
32	S	GET	1	0	GET_DESCRIPTOR	REPORT_DESCRIPTOR type	0x0000	REPORT Descriptor	42.997 ms	00017.0733 6185

Transfer	F S	Interrupt	ADDR	ENDP	Bytes Transferred	Time	Time Stamp
33	S	IN	1	1	4	8.000 ms	00017.1077 6017

Document Number: AN3560
Rev. 1
07/2008

freescale™
semiconductor