Freescale Semiconductor
Application Note

Document Number: AN3564
Rev. 1, 03/2011

Customize the USB Application Using the MC9S08JM

In-depth Understanding of the Freescale USB Stack for MC9S08JM Devices

by: Derek Liu, José Ruiz, Eduardo Viramontes
China system and applications team and RTAC Americas

Contents

1 Introduction 1
1.1 Constraints 2
2 USB Module Overview 2
3 Stack Structure 2
3.1 Main program structure 2
4 Customize Application 6
4.1 Declaration of endpoints 6
4.2 Sending and receiving data 9
5 USB Application Start-up 10
6 USB to Communication Peripherals Bridge (SPI, IIC,...)10
6.1 Application description 10
6.2 Application configuration 10
6.3 Bridge Protocol 11
6.4 How the stack is used 13

1 Introduction

USB Stack

The MC9S08JM devices are members of the Freescale Flexis™ series of microcontrollers. The Flexis series includes 8-bit and 32-bit microcontrollers that share pin-to-pin compatibility, one development tool, and same peripherals. Flexis develops a co nnection point on the Freescale Controller Continuum where 8-bit and 32-bit compatibility becomes reality.

The 8-bit MC9S08JM series MCUs are full-speed USB 2.0 devices with on-chip transceiver. They provide best-in-class USB module performance, system integration, and software support. The USB module on the MC9S08JM family MCU has seven endpoints and 256 bytes RAM.

USB-enabled devices require software to allow communication transactions between a device and host. Although the USB module incorporates several parts of the USB protocol like physical signaling and several data

© Freescale Semiconductor, Inc., 2007, 2011. All rights reserved.

filters, software needs to control these layers. This is a USB stack. It takes care of transactions like USB module configuration, USB enumeration, transaction type configuration, handling endpoints, and sending and receiving data. The stack allows the user code simple operation after configuration has been done, allowing for faster application development.

The following document describes the software structure of the Freescale USB stack for MC9S08JM devices, explains how to configure it, and guides the user in quickly developing USB applications. To get the most out of this document, you should have a basic understanding of the USB protocol.

1.1 Constraints

This document and the stack software cover the basics of transfering information with the MC9S08JM USB module. No information on higher level transfers like bulk, interrupt, or isochronous transfers + covered other than endpoint configuration.

2 USB Module Overview

JM60/32/16/8 devices have seven endpoints that can be used to build seven communication pipes between the USB host and the device.

Endpoint 0 is bidirectional (IN and OUT), and it is mainly used for control transfer. Endpoint 0 is required for all USB devices.

Endpoints 1 to 6 are unidirectional. They can only be configured to do IN or OUT direction communication at a given time.

Endpoints 5 and 6 are double-buffered, which can also be called a ping-pong buffered. One buffer can be operated by the MCU, while the other is doing communication with host, controlled by the serial interface engine (SIE). With this kind of endpoint, the communication efficiency is improved because the MCU waiting time is shortened.

The types of communnication transfers supported by the module are: control, bulk, isochronous, and interrupt transfers.

For an in-depth description of the USB module, consult application note titled *USB Device Development with the MC9S08JM60* (Document AN3560),USB device development with JM60/16.

3 Stack Structure

3.1 Main program structure

This section describes the software structure with flow diagrams.

3.1.1 System initialization

This code initializes the USB module so it is ready to connect to a USB host. The initialized components are:

- Bus clock (set to 24 MHz)
- Pull-up resistor
- Voltage regulator
- Endpoints

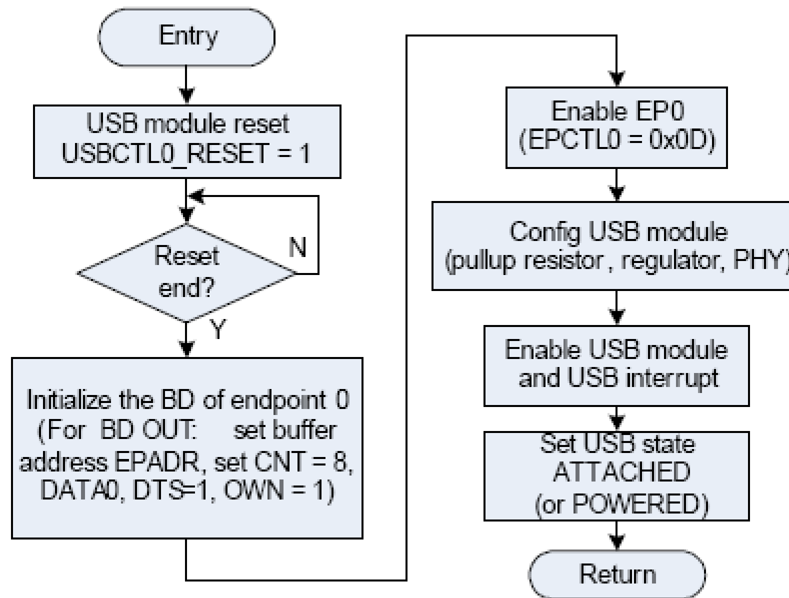

Figure 3-1. Initialization routine

The firmware sets the RESET bit in the USBCTL0 register to reset the USB module and resets all registers to their default values.

The firmware finishes the initialization of USB RAM, mainly buffer descriptor (BD) registers, especially for the endpoint 0. The BD register for endpoint 0 OUT is set. The EPAD register is set to point the endpoint buffer in USB RAM. The BC register is set to 8 for receiving the data packet with length of 8. The status and control register is set for receiving DATA0 packet (DTS =1, OWN = 1, DATA0/1 = 0).

Endpoint 0 is enabled, and the USB module is configured to enable the pull-up resistor, regulator, and PHY (Physical) according to the system hardware design.

The USB module and the USB interrupt are enabled. The USB device state is set to ATTACHED. If the USB is self-powered, the USB device state should be set to POWERED. The firmware can change to ATTACHED after detecting the USB device has been attached to the bus.

3.1.2 Main loop

The main loop calls the initialization function and then polls the USB stack for status changes. In this stage, you can monitor a reception flag to know when the reception buffer has been filled.

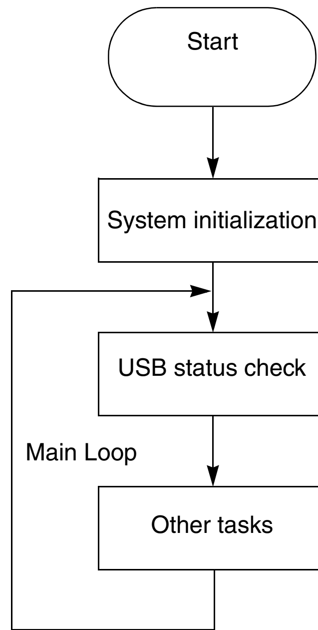

Figure 3-2. Main loop

3.1.3 USB interrupt subroutine

The following flow chart describes the USB interrupt subroutine. The USB ISR (interrup subroutine) manages USB events and some of the status changes (the other ones are managed by the USB status check in the main loop). The USB interrupt flags are checked one by one. If one interrupt flag is set and it is enabled, the ISR jumps to the associated function.

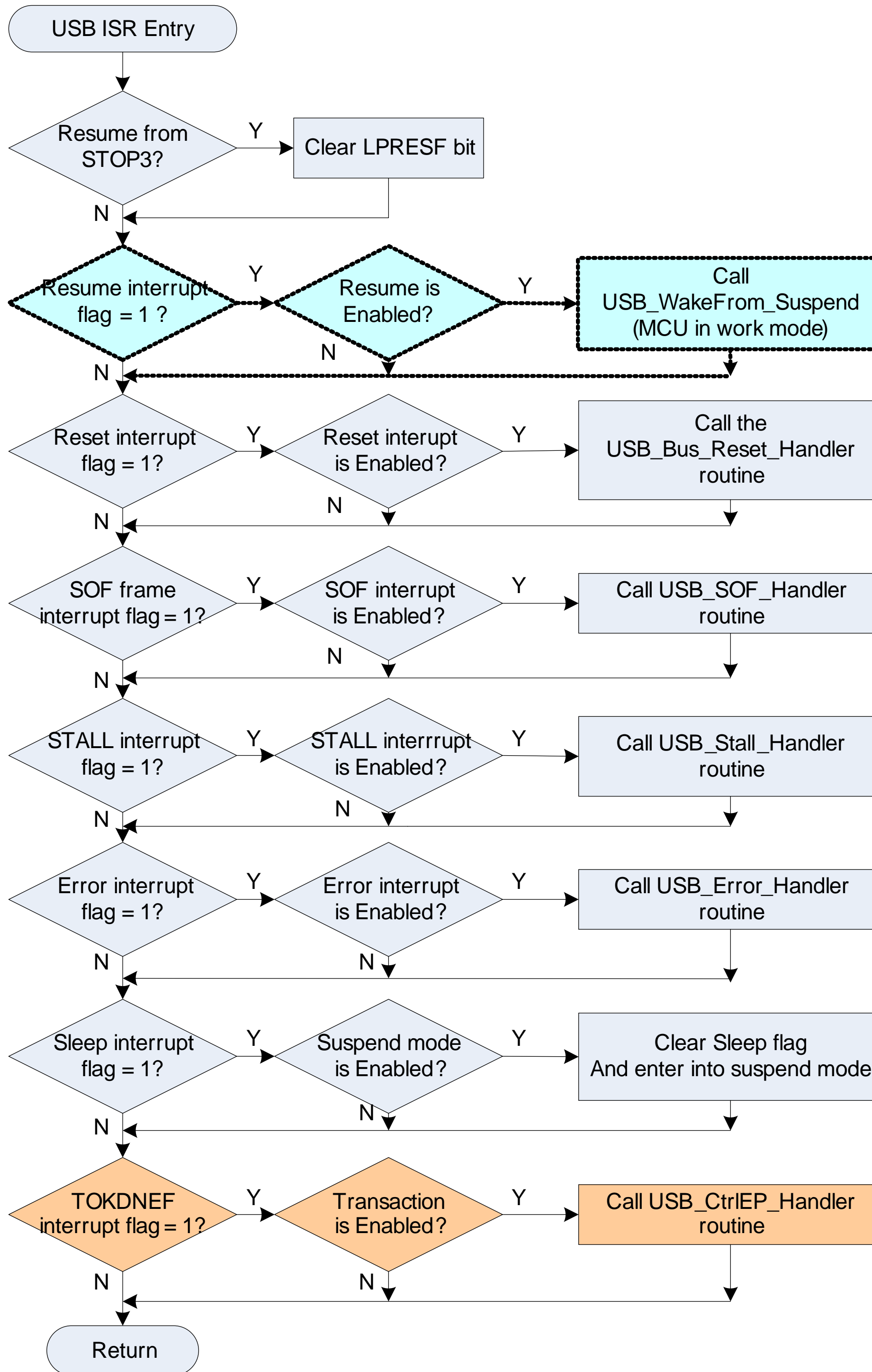

Figure 3-3. USB interrupt subroutine

4 Customize Application

The following section describes what you need to do to get your USB application running.

4.1 Declaration of endpoints

This is the critical step in customizing your application. When you declare an endpoint, you define if it is IN or OUT, the transfer types it supports (isochronous, interrupts, bulk), what endpoints work in what way, how much data each endpoint holds, etc.

4.1.1 USB descriptors

All USB devices have a hierarchy of descriptors that describe the device to the host with the following type of information:

- What the device is
- Who made it
- What version of USB it supports
- How many ways can it be configured
- The number of endpoints and their types etc.

The most common USB descriptors are:

- Device descriptor
- Configuration descriptor
- String descriptor
- Endpoint descriptor
- Interface descriptor

Each one of these provides a small description of the behavior of the device.

The descriptor entries are in file Usb_Description.c.

To customize the device, all these descriptors must be changed according to the new application.

See the Usb_Description.c file that shows the purpose of each table entry.

4.1.2 Pipes

USB communication is based on pipes (logical channels). Pipes are connections started by the HOST to a logical entity on the device named endpoint.

Each pipe (the connection to the endpoint) can be OUT (from Host to Device) or IN (from Device to Host). IN and OUT transactions are always from the point of view of the host, so OUT is always a data transfer from host to device and IN is always a data transfer from device to host.

4.1.3 USB memory access

The USB RAM memory has a 256-bytes bank shared with the USB SIE. For more information, check the application note titled *USB Device Development with the MC9S08JM60* (document AN3560). This bank starts at address 0x1860. That is, address 0x1860 (for the core) is exactly 0x00 address for the SIE. This shared memory holds all the endpoints information. The endpoint configuration process involves writing all the endpoints data for your application to this memory space.

The access to this shared memory is managed by semaphores. This means that the access to the USB buffers is fully controlled. To coordinate the USB buffers, a user buffer (outside shared memory) for each endpoint must be implemented to prevent illegal access to this locations during SIE operation.

To efficiently manage USB endpoint communications, the USB module implements a buffer descriptor table (BDT). This BDT provide status and control information for each active endpoint.

The first 32 entries in the USB RAM contain the BDT values of all endpoints or reserved. The BDT table contains the size of the next transaction (IN or OUT), the offset of the endpoint buffer in the USB RAM, and the synchronization bit (Data0 or Data1). The BDT entries in the stack code are C language structures and access to BDT is through these structures.

The SIE can only access the shared RAM in 16-byte mode. Each endpoint buffer must be 16-byte aligned even if the buffer size is only 8 bytes. For example, the first two buffers are regularly for the Endpoint 0 (EP0) IN and OUT and the first 32 bytes are for the BDT table, so EP0 IN starts in 0x20 (SIE) or 0x1880 (Core). EP0 OUT starts at 0x30 (SIE) or 0x1890 (Core). The same principle applies for each user endpoint. If EP1 is 8 or 16 bytes length, its buffer starts at 0x40 (SIE) or 0x18A0 (core) and EP2 starts at 0x50 (SIE) or 0x18B0 (core).

The base address declaration of the endpoints' buffers must be right shifted two times and stored in the BDT offset register called EPx_Set.Addr (where x is the number of endpoint).

For example, the EP1 buffer starts at 0x40 of the SIE (0x18A0 Core).

1. First, declare the Endpoint buffers

```
/* User Buffer for Enpoint 1 (User space) */
byte EP1_Buffer[EP_MAX_SIZE];
/* Endpoint 1 buffer (Shared space) */
byte EP1_BDT_Buffer[EP_MAX_SIZE] @0x18A0;
```

2. Next, Initialize the Base address in the BDT structure

```
SIE address = EP1_Set.Addr = (0x40) >>2 = 0x10
/* Size of next transaction (this field is updated in each transaction)*/
EP1_Set.Cnt = 0x00;
/* Base address of Endpoint buffer */
EP1_Set.Addr = 0x10;
/* Semaphores for buffer access and sync signals */
        EP1_Set.Stat._byte = _SIE|_DATA0|_DTS;
```

4.1.4 Enabling endpoints

To enable each endpoint, write the direction and handshake configuration.

```
EPCTLx= EP_IN|HSHK_EN; in case of IN Endpoint and handshake enabled
EPCTLx= EP_OUT|HSHK_EN; in case of OUT Endpoint and handshake enabled
```

4.1.5 Enumeration process

The enumeration process is the set of operations that occur immediately after the host has detected a device has been connected to the USB bus and has (by hardware) determined the communication speed. Enumeration consists in the host asking the device (the MC9S08JM in this case) for its configuration and assigning a specific address. Configuration from the device point of view is to send the host information on all endpoints, buffer sizes, and types of tranfers (interrupt, bulk, and so on.). This enables the host to communicate with the device in an expected manner. All communications in this stage occur through endpoint 0, also known as the control endpoint.

Stack code is prepared to manage the enumeration process so that if the endpoint has been properly configured, stack code takes care of the complete process. This uses no more code than calling the initialization routine Initialize_USBModule() and polling the Check_USBBus_Status() in the infinite loop of the main program to assure proper refreshing of the USB communications status.

The USB communication status in the stack follows a state machine that runs from first detection of physical connection to a configured state in which the device can be used. The following image describes this state machine.

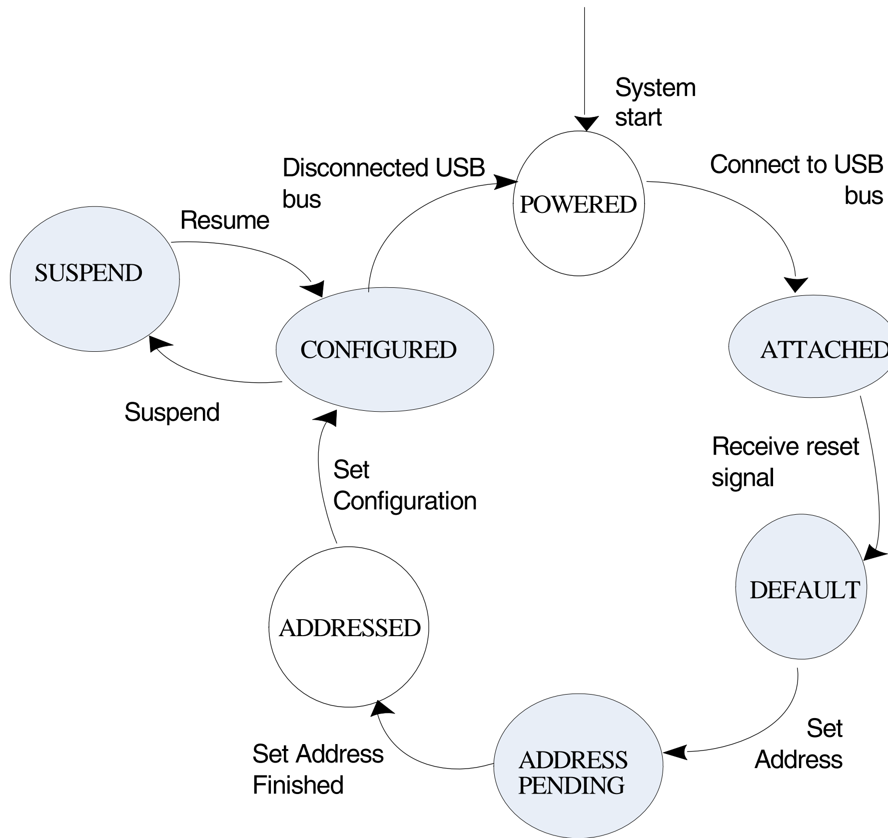

Figure 4-4. USB Communication state machine

The function Check_USBBus_Status() with interrupt driven events refresh the status as shown in the image. This means the USB device can be used when it is in the CONFIGURED state. Values for the status are refreshed in the variable Usb_Device_State.

4.2 Sending and receiving data

After endpoints have been properly configured, send and receive functions can be used to communicate with the host device. Check the status of the communications before calling any code to know if the USB module has been enumerated by the host. As mentioned earlier, the Usb_Device_State variable holds the current state of USB communications. Check_USBBus_Status() and others called by interrupts to update the status. Data should only be sent when Usb_Device_State holds the value CONFIGURED because this means the USB host has recognized the USB device configuration.

4.2.1 How to send data

Only endpoints configured as IN are capable of sending data (everything is named from the host side, so an IN endpoint sends data into the host). The way the USB stack sends data is by writing data into the endpoint buffer, defining the number of datum being sent, and granting the serial interface engine control of that particular endpoint buffer. Next time it receives an IN token, it sends the data and generates an interrupt when the transaction is finished. This is summarized in the following way:

- Write data to the endpoint buffer being used: EPx_Buffer[n] (where x is the endpoint number and n is the array size).
- Call the IN endpoint API function: void EndPoint_IN(UINT8 u8Endp,UINT8 u8EPSize). The arguments are: u8Endp is the endpoint number, the file USB_User_API.h holds the definitions for these endpoint numbers, and u8EPSize is the number of bytes sent.
  - Use the following macros to send in the u8Endp argument:
    - EP0 for endpoint 0
    - EP1 for endpoint 1
    - EP2 for endpoint 2
    - EP3 for endpoint 3
    - EP4 for endpoint 4

This process works on any endpoint, one to four, that has been configured as an IN endpoint.

4.2.2 How to read received data

The USB stack has been written to use interrupts. When an endpoint has been configured as an OUT endpoint and data is sent to it, an interrupt is generated and the stack refreshes a flag that tells the user API there is data ready in the endpoint. The function CheckEndPointOUT(UINT8 u8Endp) returns a 1 when there is data in the endpoint and 0 if there is no data available. As before, u8Endp is the endpoint number and the definitions in USB_User_API.h should be used.

This process works on any endpoint, one to four, that has been configured as OUT endpoint.

5 USB Application Start-up

1. Configure the endpoints.
   a) Write the BDT table for each endpoint needed.
   b) Determine the transaction type.
   c) Determine your vendor and product IDs.
   d) Write the endpoint descriptors in Usb_description.c. For more information on endpoint descriptors, consult the USB Standard revision 2.0 on www.usb.org.
2. Check Usb_Device_State for the CONFIGURED state.
3. If USB has been enumerated (is configured), start checking OUT endpoints for data or sending data through IN endpoints.

6 USB to Communication Peripherals Bridge (SPI, IIC, and SCI)

6.1 Application description

A communications bridge is an application that takes input from one communications port and sends it through another. In this case, a bridge is used to connect data from the USB to one of three selected peripherals available in the MC9S08JM:

- Serial peripheral interface (SPI),
- Inter-integrated circuit communication (IIC)
- Serial communications interface (SCI).

To add flexibility to the application, a protocol has been defined to allow configuration of the selected peripherals. To simplify this configuration capability, only the most important parts of the MC9S08JM peripherals are available (for example, the configuration options for the SCI module where selected to act like a Windows® COM port). All three modules are initialy configured to specific values so that, if required, data can be sent through the bridges without any configuration.

A Windows GUI connected through Windows WinUSB drivers provides the user with the ability of configurating the MC9S08JM peripherals and sending and receiving data through the bridges.

6.2 Application configuration

The USB module has been configured to work as a vendor specific device. The WinUSB drivers use a specific vendor ID with which the PC drivers are able to communicate. This configuration is written in the Usb_Descriptor.c file. Four endpoints are used (besides Endpoint 0, USB protocol already required this). All endpoints are bulk transfer based; two are 16-bytes long and two are 32-bytes long. Endpoint 1 is an OUT endpoint. Commands from the host are sent to the MC9S08JM through this pipe. Endpoint 2 is an IN endpoint. Status replies to commands are sent to the host from this endpoint. Endpoints 3 and 4 are the 32 byte data pipes. Endpoint 3 receives data from the host and sends it to the configured peripheral. Endpoint 4 takes data from the peripherals and sends it to the host.

6.3 Bridge protocol

A protocol configures the MC9S08JM peripherals. This allows the application operational flexibility. The data format in the command and status pipes is:

| 1 byte: peripheral ID | 1 byte: command | 14 bytes: data | *Eqn. 6-1*

Because this application example uses three communications peripherals, three different IDs are used:

- 0x03 for SPI
- 0x04 for IIC
- 0x05 for SCI

The following tables describe the commands that are used.

Table 6-1. SPI – Host to device

Pipe	Command	Description	Data
Endpoint 3	0x00	Data transfer to SPI bus	Application data (up to 31 bytes)
Endpoint 1	0x01	Defines the baud rate for transmission of SPI port. Baud rate is limited to the device capability to generate the required baud rate. Any baud rate that cannot be exactly generated is generated to the closest attainable value. In SPI baud rate is the frequency of the serial clock.	2 bytes Byte 1 – Baud rate prescaler divisor. Where maximum value is 7, and prescaler value equals Byte 1 value + 1. Check Table SPI Baud Rate Prescales Divisor of the MC9S08JM60 data sheet. Byte 2 – Baud rate divisor. Check Table SPI Baud Rate Divisor of the MC9S08JM60 data sheet. Formula: Baud rate = Bus rate (24 Mhz)/(Byte 1 * Byte 2)
Endpoint 1	0x02	8 or 16 bit transmission	1 byte, 0 for 8 bits, 1 for 16 bits
Endpoint 1	0x03	Master or slave	1 byte, 0 for master, 1 for slave
Endpoint 1	0x04	Full-duplex or half-duplex (in SPI half duplex also means 1 wire to transmit and receive)	1 byte, 0 for full-duplex, 1 for half-duplex/1 wire
Endpoint 1	0x05	Clock phase	1 byte, 0 first edge on serial clock occurs at the middle of the first cycle of a data transfer 1 first edge on serial clock occurs at the start of the first cycle of a data transfer
Endpoint 1	0x06	Clock polarity	1 byte, 0 active-high SPI clock (idles low), 1 active-low SPI clock (idles high)
Endpoint 1	0x07	Shifter direction or significant bit transfer start	1 byte, 0 for most significant bit first, 1 for least significant bit first

Table 6-2. Device to host

Pipe	Command	Description	Data
Endpoint 4	0x00	Data transfer from SPI bus	Application data (up to 31 bytes)
Endpoint 2	0x01	Baud rate ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x02	8 or 16 bit transmission ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x03	Master or slave configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x04	Full duplex or half duplex configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x05	Clock phase configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x06	Clock polarity configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x07	Shifter direction or significant bit transfer start configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted

Table 6-3. IIC – Host to device

Pipe	Command	Description	Data
Endpoint 3	0x00	Data transfer to IIC bus	Application data (up to 31 bytes)
Endpoint 1	0x01	Baud rate configuration	2 bytes: Byte 1 – Baud rate multiplier, where: 0 -> multiply by 1 1 -> multiply by 2 2 -> multiply by 4 Byte 2 – Baud rate divider, where table IIC Divider and Hold Values of the MC9S08JM60 data sheet defines the divider values and the formula for these values is: Baud rate = bus rate (24 MHz)/(multiplier * divider)
Endpoint 1	0x02	Master slave configuration	0x00 for Slave 0x01 for Master
Endpoint 1	0x03	Slave address configuration – If master, this is the address where data is transferred. If slave, this is the address configured as a slave address in the IIC module.	Byte 1 – 0 for 7 bit addres, 1 for 10 bit address. Byte 2 – 7 bit address or low part of 10 bit address. Byte 3 – Highest three bits of 10 bit address (shifted to the right of the byte)

Table 6-4. Device to host

Pipe	Command	Description	Data
Endpoint 4	0x00	Data transfer from IIC bus	Application data (up to 31 bytes)
Endpoint 2	0x01	Baud rate configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x02	Master slave configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x03	Slave address configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted

Table 6-5. SCI – Host to device

Pipe	Command	Description	Data
Endpoint 3	0x00	Data transfer to SCI port	Application data (up to 31 bytes)
Endpoint 1	0x01	Baud rate configuration	2 Bytes Byte 1 – High part of baud rate divider, where maximum value is 31. Byte 2 – Low part of baud rate divider. Formula: Baud rate = Bus rate (24 MHz) / (16 * baud rate divider)
Endpoint 1	0x03	Parity configuration	0x00 – None 0x01 – Odd 0x02 – Even

Table 6-6. Device to host

Pipe	Command	Description	Data
Endpoint 4	0x00	Data transfer from SCI port	Application data (up to 31 bytes)
Endpoint 2	0x01	Baud rate configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted
Endpoint 2	0x03	Parity configuration ACK	0xFF – configuration accepted 0x00 – configuration not accepted

The PC GUI provided with this application note selects configuration and sends and receives data.

6.4 How the stack is used

This application uses the functions EndPoint_IN(UINT8 u8Endp,UINT8 u8EPSize) and CheckEndPointOUT(UINT8 u8Endp) to use the stack. Explained in the application configuration, the endpoints are used as vendor specific endpoints. The USB stack user functions are part of the USB_User_API.c file. These functions are managed by the Protocol_Handler.c file that communicates with the bridge.c file. The application is layered, making it easy to port or change it. The layer model is shown in Figure 5.

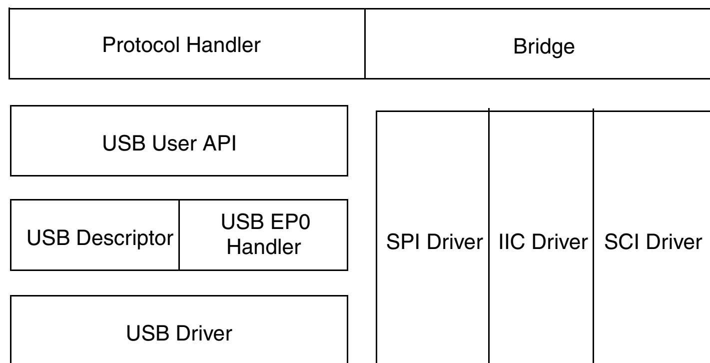

Figure 6-5. Application layer model

As can be seen, this model makes it easy to add functionality to a user application. The layers labeled as bridge and protocol handler can be any other application level code that uses the USB code. The SPI, IIC, or SCI layers can be changed for any other hardware level drivers like an analog-to-digital converter, a PWM output, or even a display.

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

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

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3564
Rev. 1
03/2011

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© Freescale Semiconductor, Inc. 2007, 2011. All rights reserved.

freescale™
semiconductor