Freescale MQX™ RTOS USB Device User's Guide

MQXUSBDEVUG
Rev. 4
02/2014

freescale™

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXUSBDEVUG
Rev. 4, 02/2014

freescale™

Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, see freescale.com and navigate to Design Resources>Software and Tools>AllSoftware and Tools>Freescale MQX Software Solutions.

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 1	12/2011	Initial Release coming with MQX RTOS version 3.8
Rev. 2	06/2012	Updated for MQX RTOS version 3.8.1 - see 5.5, “USB DDK Changes in MQX RTOS version 3.8.1
Rev. 3	04/2013	Minor edits.
Rev. 4	10/2013	Updated content to reflect the switch from MQX types to C99 types.

Chapter 1 Before You Begin

Chapter 2 Getting Familiar

Chapter 3 Freescale MQX USB Device Stack Architecture

Chapter 4 Developing new Class Drivers

Chapter 5 Developing new Applications

Appendix A Working with the Software

Appendix B Human Interface Device (HID) Demo

Appendix C Virtual Communication (COM) Demo

Appendix D MSD Demo

Chapter 1 Before You Begin

1.1 About the Freescale MQX™ USB Device Stack

The Universal Serial Bus commonly known as USB is a serial bus protocol that can be used to connect external devices to the host computer. Currently, it is one of the most popular interfaces connecting devices such as microphone, keyboards, storage devices, cameras, printers, and many more. USB interconnects are also getting more and more popular in the medical segments. The Freescale MQX USB Stack enables you to use Freescale 16-bit and 32-bit MCUs, for example: Kinetis, CFV1, CFV2 and so on, silicon to make the devices listed above.

It abstracts the details of Kinetis, CFV1, and CFV2 devices. It provides a higher level interface to the application. The application developers only need to concentrate on the application at hand without needing to worry about the USB details.

1.2 About This Book

This book describes how to use the Freescale MQX USB Stack. This book does not distinguish between USB 1.1 and USB 2.0 information unless there is a difference between the two.

This book contains the following topics:

Table 1-1. MQXUSBDEVUG summary

Chapter Tittle	Description
Before you begin	This chapter provides the prerequisites of reading this book.
Getting Familiar	This chapter provides the information about the Freescale MQX USB Device Stack software suite.
Freescale MQX USB Device Stack Architecture	This chapter discusses the architecture design of the USB stack suite.
Developing new Class Drivers	This chapter discusses the steps that a developer should take to develop a new USB device class driver.
Developing new applications	This chapter discusses the steps that a developer should take to develop applications on top of the pre-developed classes.
Working with the Software	This chapter provides information on how to build, run, and debug drivers and applications.
Virtual Communication (COM) Demo	This chapter provides the setup and running Communication Device Class (CDC) demo using USB stack where Kinetis, CFV1, and CFV2 devices are used as examples.
MSD Demo	This chapter provides the setup and running Mass Storage Class (MSC) demo using USB stack where Kinetis, CFV1, and CFV2 devices are used as examples.
Human Interface Device (HID) Demo	This chapter provides the setup and running HID demo using USB stack where Kinetis k40, CFV1, and CFV2 devices are used as examples.

1.3 Reference Material

See the following reference material for more details:

- *Universal Serial Bus Specification Revision 1.1*
- *Universal Serial Bus Specification Revision 2.0*
- *Freescale MQX™ RTOS USB Device API Reference Manual (MQXUSBDEVAPI)*

Chapter 2 Getting Familiar

2.1 Introduction

Freescale Semiconductor provides Freescale MQX USB Device Stack operating at both low-speed and full speed. Freescale MQX RTOS comes with support for a complete software stack combined with basic core drivers, class drivers, and sample programs that can be used to achieve the desired target product. This document intends to help customers develop an understanding of the USB stack in addition to providing useful information about developing and debugging USB applications. A good understanding of the USB stack can be developed when this document is read in combination with API documentation provided with the software suite. The document is targeted for firmware and software engineers who are familiar with basic USB terms and are directly involved developing the product on Freescale MQX USB Device Stack.

2.2 Software Suite

The software suite comprises of the USB low level drivers for the Kinetis, CFV1, and CFV2 families, generic class drivers, and applications. The class drivers are programmed with generic code. Therefore, the user application can be used with other processors such as CFV1 and CFV2 without any code change, provided the low level drivers comply with the driver interface.

2.2.1 Directory Structure

The following bullets display the complete tree of Freescale MQX USB Device Stack. If a customer makes a device-only product, the following directories are the only cause for potential concern.

- Freescale MQX RTOS \usb\device\examples - Sample codes for device firmware development.
- Freescale MQX RTOS \usb\device\source - All source files that drive the device controller core.
- Freescale MQX RTOS \lib - Output where all libraries and header files are copied when the USB software tree is built.

Table below briefly explains the purpose for each of the directories in the source tree.

Table 2-1. Directory Structure

Directory Name	Purpose
usb	Root directory of the software stack.
usb\device\build	Project files to build USB device stack library for different platforms and tool chains.
\lib\usbd	Output directory of the build.
usb\device\source	USB Device Stack source code.
usb\device\examples	Device firmware development examples.
usb\device\source\classes	C source file for class layer.
usb\device\source\device	C source file for low level driver layer.

Table 2-1. Directory Structure (continued)

Directory Name	Purpose
usb\device\source\include	Header files.
usb\device\source\rtos	RTOS layer source.

Figure below shows the default directory structure.

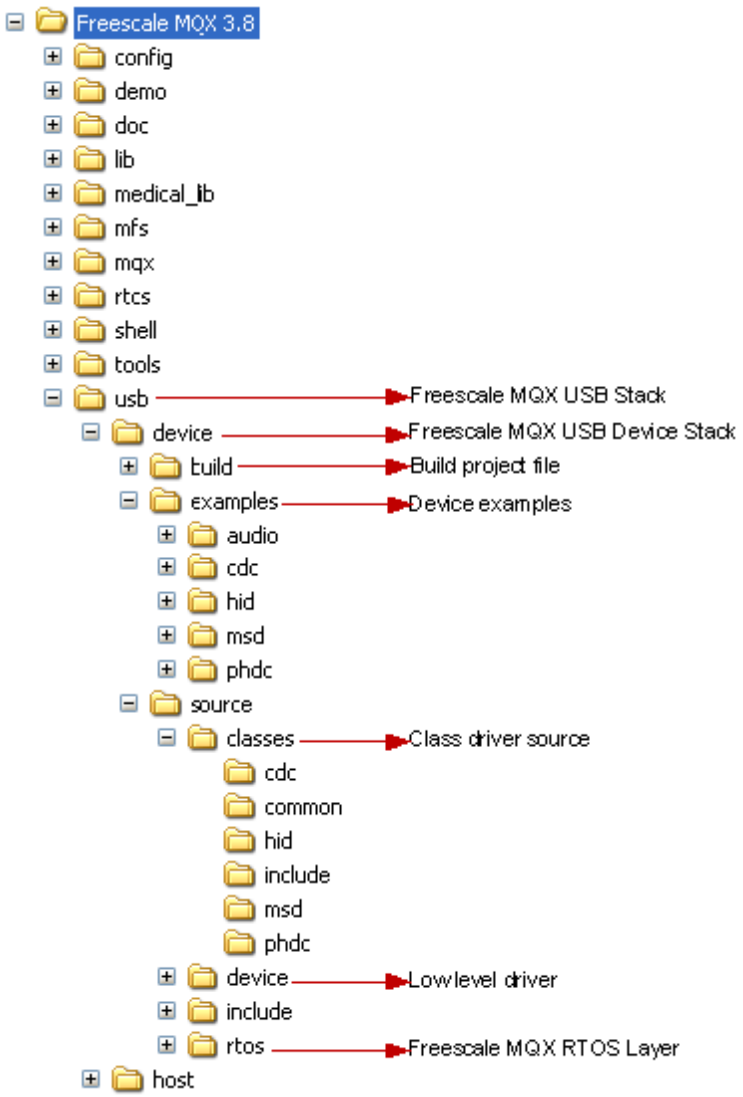

Figure 2-1. Default Directory Structure

Chapter 3 Freescale MQX USB Device Stack Architecture

3.1 Architecture Overview

Figure below shows the Freescale MQX USB Device Stack architecture.

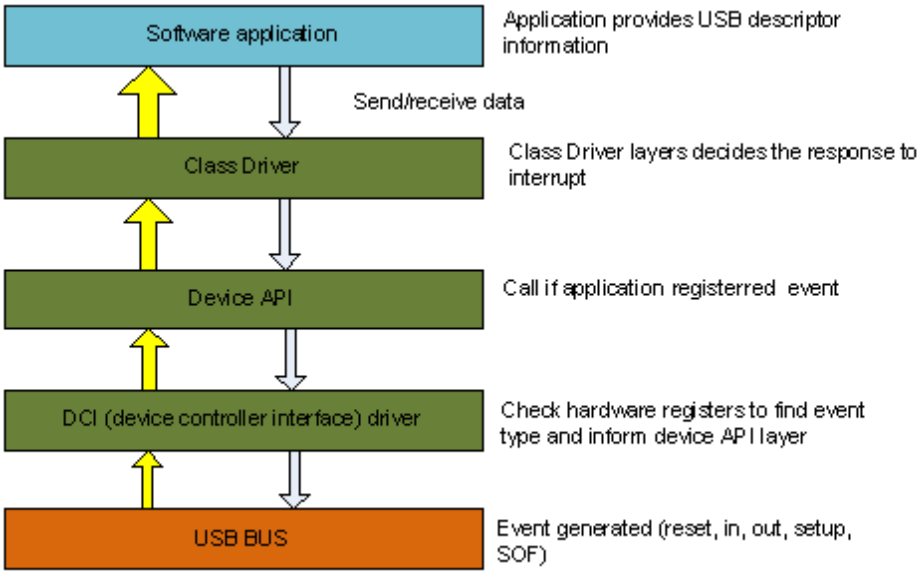

Figure 3-1. MQX USB Device Stack architecture

Freescale MQX USB Device stack supports low-speed and full-speed device cores. By definition, a device application is a responder to the host commands. This is a simpler application in comparison to host driver applications.

The MQX USB Device stack provides Device API layer that is abstraction to the hardware. Therefore, the application firmware can be written without the need to worry about a specific hardware core. To explore the device API design, trace the API routines to the hardware. The following is a summary of the functions that are implemented by this layer.

- Registration and un-registration of services with the USB Device stack.
- Registration and control of endpoints behavior. For example, stalling an endpoint.
- Sending and receiving data buffers to the specific endpoints.
- Providing status of a specific transfer on an endpoint.

The class driver layers implement various class drivers that have different functions. The USB Chapter 9 requests are also part of the functionality of the class driver layer. These are implemented as a common module and can be used as is to develop new classes. Some of the examples provided here are storage, human interface device, communication device, and so on.

Conceptually, all transfers and events from the USB bus generate hardware interrupts. Hardware drivers call the API layer, which then calls up the application if an application has registered for the event or has initialized the endpoint.

3.2 Software flows

This section describes the execution flow of the stack across various layers and modules.

3.2.1 Initialization flow

Figure below represents the stack initialization flow.

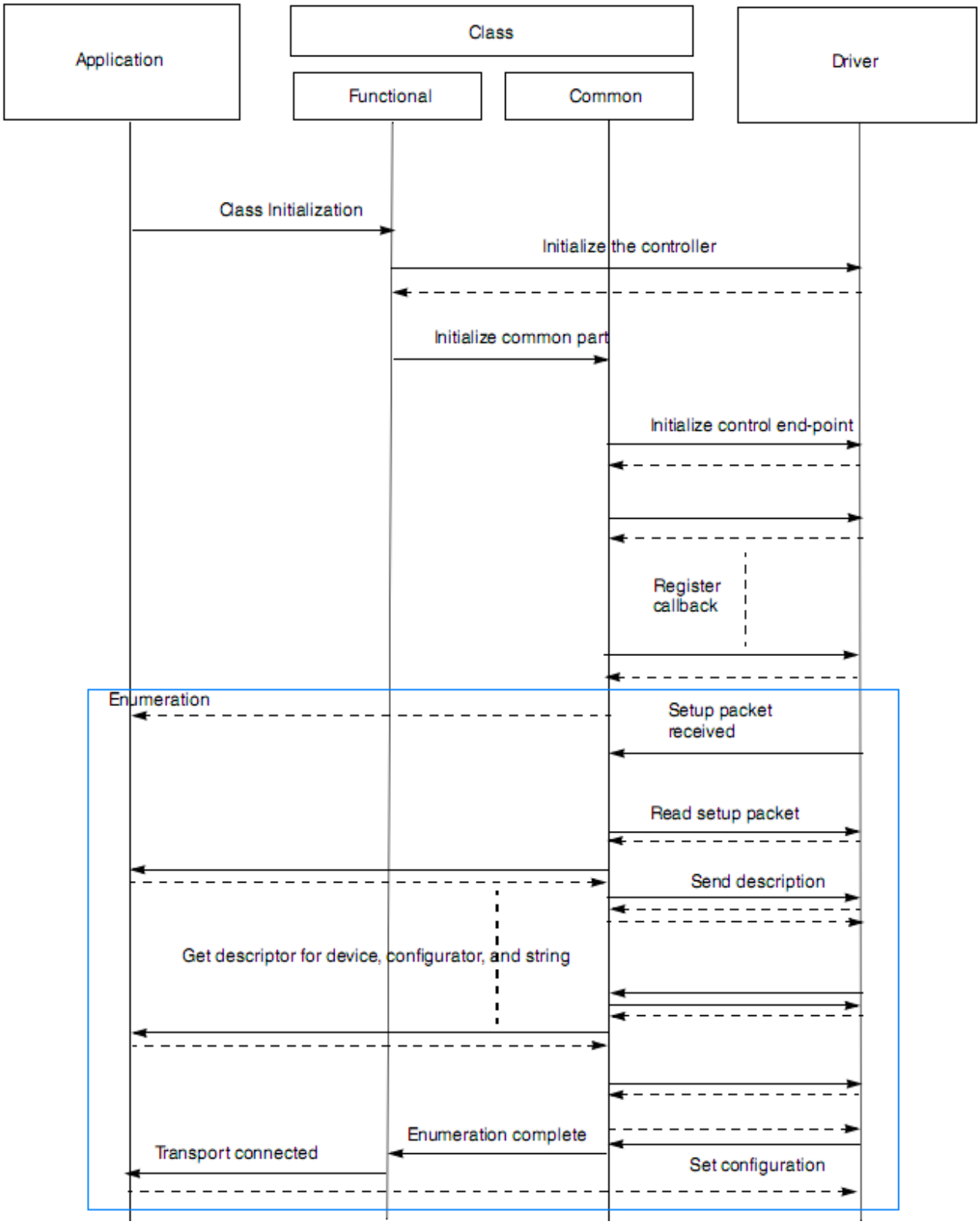

Figure 3-2. Sequence diagram for stack initiation

The initialization flow starts when the application initializes the class driver that in turn initializes the low level driver and the controller. The class driver also registers the callbacks it requires for events occurring in the USB bus. Sometime after this, the host starts the enumeration process by sending the setup packet to get the descriptors for the device, configuration, and string. These requests are handled by the class driver that uses the descriptors defined by the application. The enumeration finally ends when the host sets the device configuration. At this point, the class driver notifies the application that the connection has been established.

3.2.2 Transmission flow

Figure below represents the stack transmission flow.

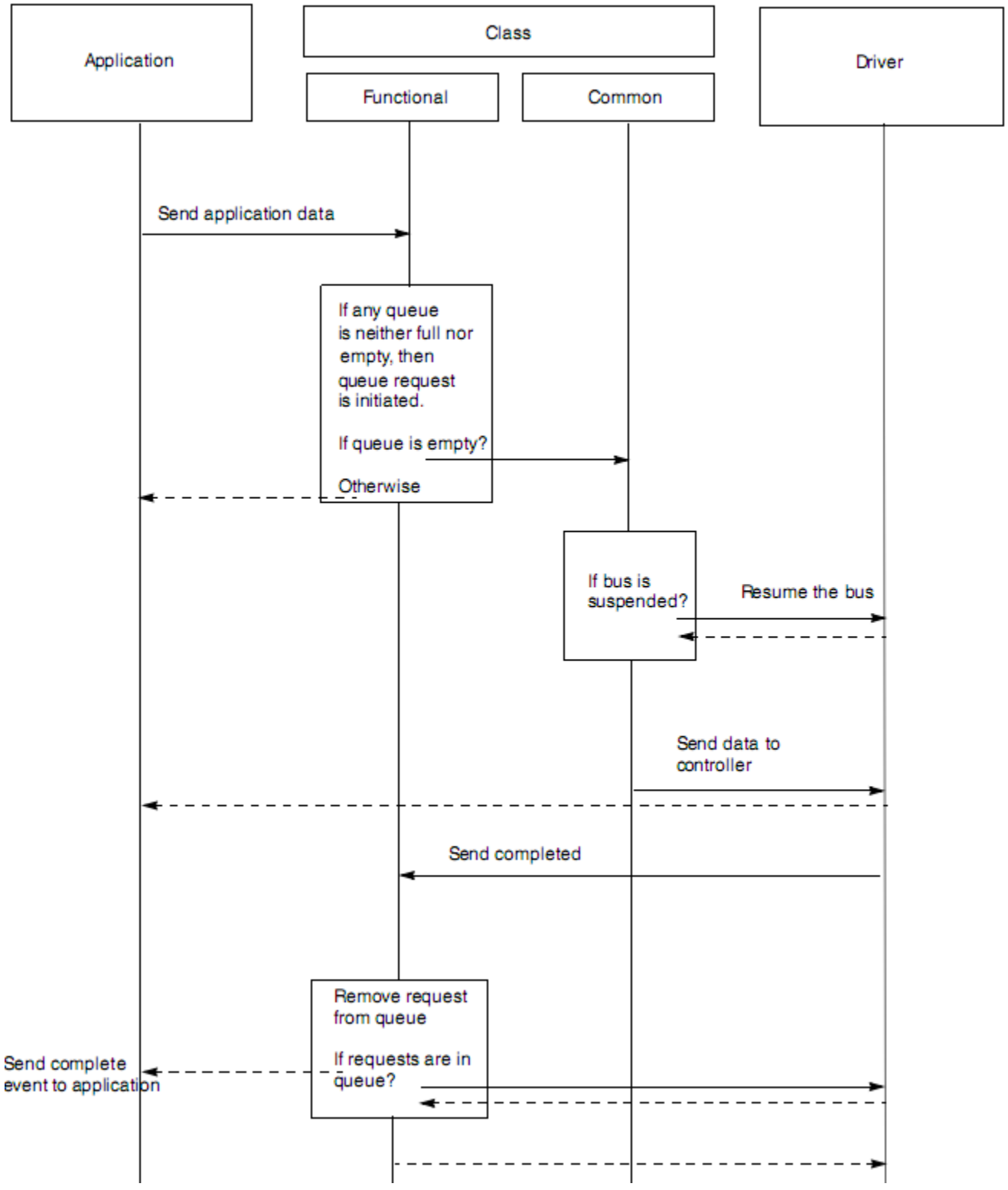

Figure 3-3. Sequence diagram for stack transmission

The application transmits data to the USB host by calling the class specific driver send API. The class driver checks for the current status of the queue. If the queue is full, the function returns a BUSY status. If there is already an outstanding request that has not been completed yet, it queues the request. Unless the queue is empty, it prepares to pass the request to the low level driver. As part of the preparation, it checks whether the bus is suspended or not. If the bus is suspended, it wakes the bus and the bus then sends the request to the lower layer driver. When send API operation is completed, the class driver removes the request from the queue and sends the next in queue if it exists.

3.2.3 Reception flow

Figure below represents stack reception flow.

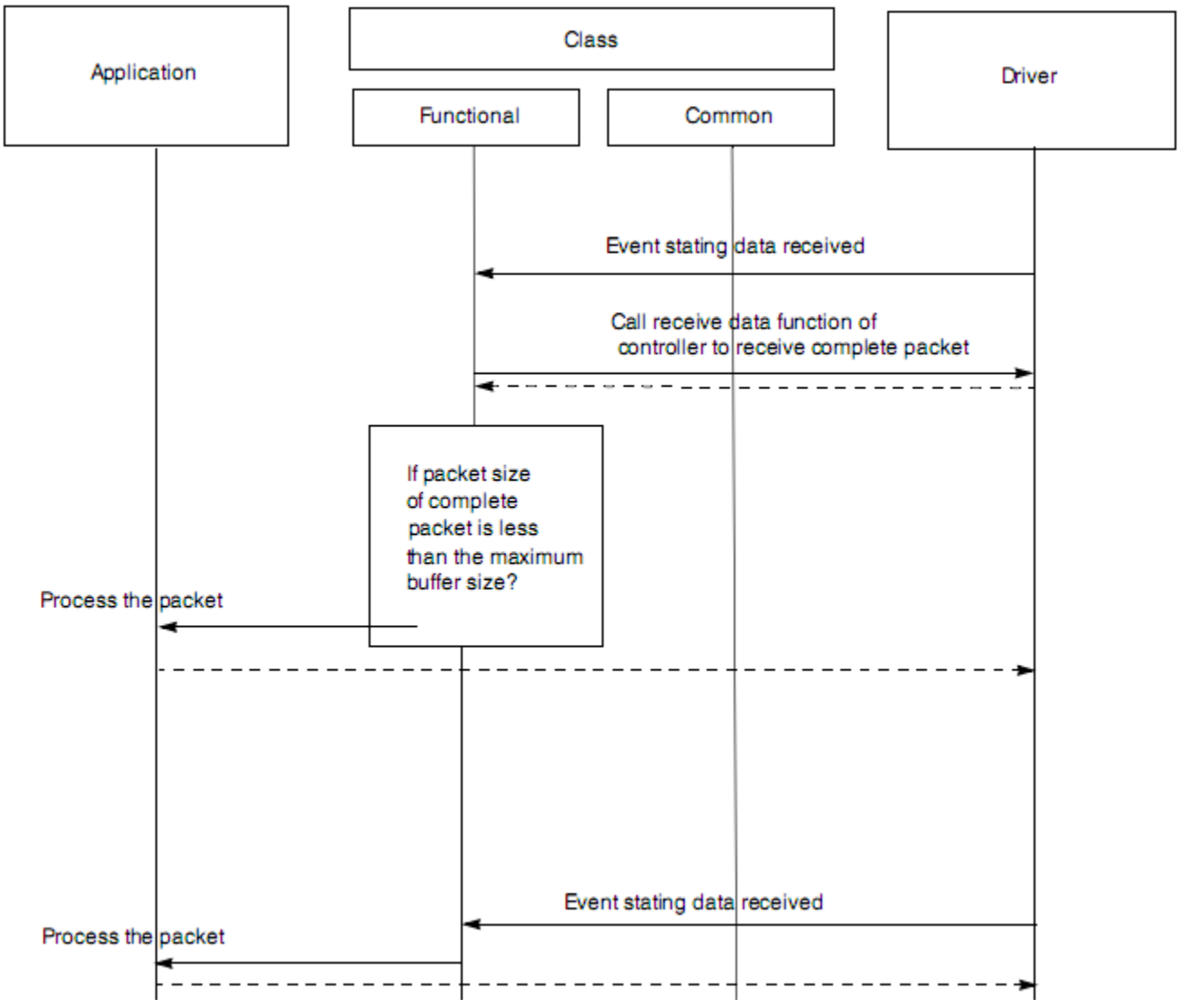

Figure 3-4. Sequence diagram for stack reception

When the controller receives the data on its receiver port, the low level driver sends an event to the class driver. The class driver calls the low level driver to receive the packet in its buffers. If the size of the packet is smaller than the size of the end-point buffer, the class driver processes the packet immediately. If the size of the packet is greater than the endpoint buffer size, the class driver waits for an event from the low level driver with the complete packet. The class driver processes the packet after it is received.

Chapter 4 Developing new Class Drivers

4.1 Introduction

This chapter provides the user methodology for developing a new class driver by using USB low level stack framework.

HID, CDC, and MSC class drivers are already implemented in the package. See the *Freescale MQX™ RTOS USB Device API Reference Manual* (MQXUSBDEVAPI) for more information.

Before starting with the development of the class drivers, see figure below for the current design of the class drivers.

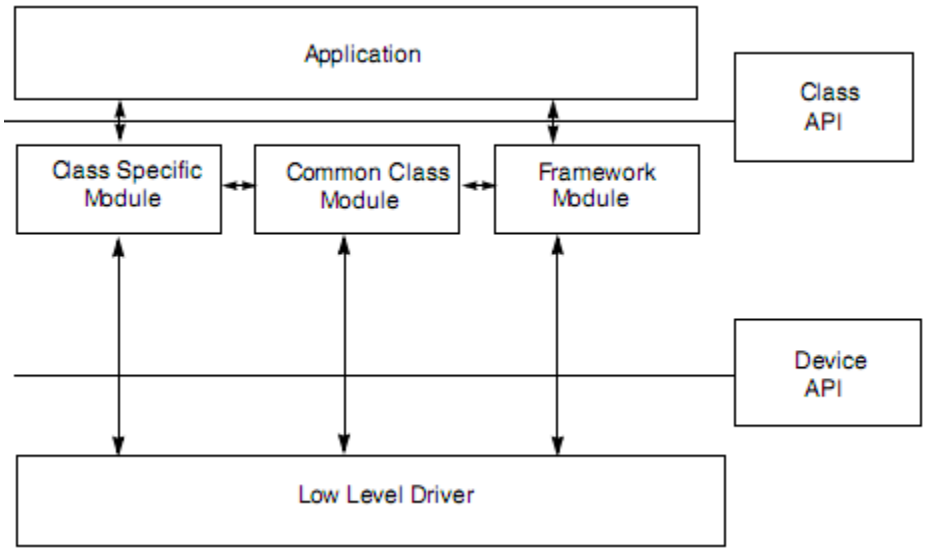

Figure 4-1. Current design of class drivers

The class driver layer is divided into three modules as follows:

- Framework Module - The framework module handles all requests to the control end point. It implements all responses to the USB Chapter 9 requests. It interacts with the application to get the USB descriptor information.
- Common Class Module - The common class module contains implementation independent of the application specific classes. It handles functions such as suspend/resume, reset, stall, and SOF that need to be present for all classes.
- Class Specific Module - This module implements class specific functionality. It implements all interactions with non control end points. The data sent and received on these end points is class specific. This module also implements the class specific requests on the control end point.

Developers can start creating a new class by using the device API and implementing complete functionality. However, it is recommended to use a design similar to the existing classes and reusing some pre-existing common modules such as the common class module and the framework module.

4.2 Steps for developing new class drivers

Follow the steps below to develop a new class driver by using an existing USB low level device framework and common class specific module.

4.2.1 Create new C header and source files for the class

First, create a file namely usb_<newclass>.h in the <include> folder as shown in the figure below.

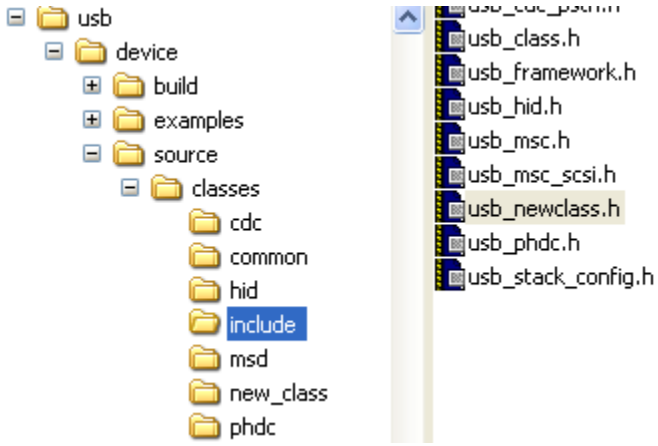

Figure 4-2. Create header file for new class

Second, create a new folder <new_class> and a file namely usb_<newclass>.c as shown in the following figure:

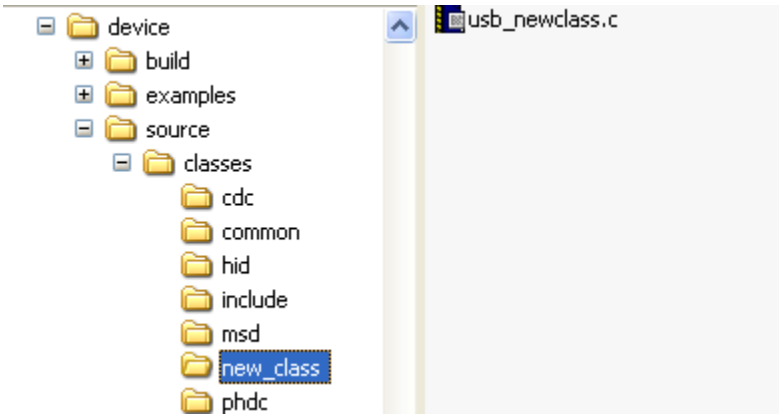

Figure 4-3. Create source file for new class

The class specific code is implemented in the .c file and the interface definition is placed into the .h file. The application must use this interface to call the class driver.

4.2.2 Preparing class device structure definition

This structure contains the device state information. It's instance is created and initialized in the class initialization function.

Pseudo Code:

```
typedef struct XYZ_device_struct
{
     _usb_device_handle            handle;
     XYZ_HANDLE                    XYX_handle;
     USB_CLASS_HANDLE              class_handle;
     USB_ENDPOINTS                 *ep_desc_data;
     <Class specific structure fields >
     USB_CLASS_CALLBACK_STRUCT     hid_class_callback;
     USB_REQ_CALLBACK_STRUCT       vendor_req_callback;
     USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT param_callback;
     USB_CLASS_XYX_ENDPOINT_DATA   XYX_endpoint_data;
}XYX_DEVICE_STRUCT,  * XYX_DEVICE_STRUCT_PTR;
```

4.2.3 Preparing class configuration structure definition

This structure holds class configuration information to be passed by the application.

Pseudo Code:

```
typedef struct XYZ_config_struct
{
     uint32_t                           desc_endpoint_cnt;
     USB_ENDPOINTS                      *ep_desc_data;
     USB_CLASS_XYZ_ENDPOINT             *ep;
     <Class specific structure fields >
     USB_CLASS_CALLBACK_STRUCT          XYZ_class_callback;
     USB_REQ_CALLBACK_STRUCT            vendor_req_callback;
     USB_CLASS_SPECIFIC_HANDLER_CALLBACK_STRUCT param_callback;
     DESC_CALLBACK_FUNCTIONS_STRUCT_PTR  desc_callback_ptr;
}XYZ_CONFIG_STRUCT, * XYZ_CONFIG_STRUCT_PTR;
```

4.2.4 Implement class initialization function

This function initializes class specific data structures. This function also initializes the USB Common Class Module and USB Device Framework through USB_Class_Init() and USB_Device_Init() respectively.

Typically, three callbacks are provided by an application for interfacing with the class driver.

- Class callback (to receive various USB bus events)
- Support for vendor specific requests.
- Handling class specific requests (application specific like string descriptor handling etc.).

Pseudo Code:

```
uint8_t USB_Class_XYZ_Init
(
```

```
      XYZ_CONFIG_STRUCT_PTR config_ptr
)
{
      uint8_t index;
      uint8_t error;
      XYZ_HANDLE handle;
      XYZ_DEVICE_STRUCT_PTR  devicePtr = NULL;

      devicePtr =
      (XYZ_DEVICE_STRUCT_PTR)USB_mem_alloc_zero(sizeof(XYZ_DEVICE_STRUCT));
      handle = USB_XYZ_Allocate_Handle();

      /* Initialize the device layer */
      error = _usb_device_init(USBCFG_DEFAULT_DEVICE_CONTROLLER,
      &(devicePtr->handle),
      (uint8_t)(config_ptr->ep_desc_data->count+1));

      config_ptr->desc_callback_ptr->handle = handle;

      /* Initialize the generic class functions */
      devicePtr->class_handle = USB_Class_Init(devicePtr->handle,
      USB_Class_XYX_Event, USB_XYZ_Other_Requests,(void *)devicePtr,
      config_ptr->desc_callback_ptr);

      <Class Specific Initialization code goes here >

      /* save the XYZ class callback pointer */
      USB_mem_copy(&config_ptr->XYZ_class_callback,
      &devicePtr->XYZ_class_callback,sizeof(USB_CLASS_CALLBACK_STRUCT));

      /* save the vendor request callback pointer */
      USB_mem_copy(&config_ptr->vendor_req_callback,
      &devicePtr->vendor_req_callback,sizeof(USB_REQ_CALLBACK_STRUCT));

      /* Save the callback to ask application for class specific params*/
      USB_mem_copy(&config_ptr->param_callback,
      &devicePtr->param_callback.callback ,sizeof(USB_REQ_CALLBACK_STRUCT));

      return error;
}
```

4.2.5 Implement class callback routine

This routine is called by USB Common Class Module to notify the class driver about various USB events. The following events are notified:

- **USB Bus Reset**:

  This event is notified when USB Bus Reset is detected by the Device Controller. The class driver should reset its data structure after receiving this event. Depending on the class requirement, the event can be propagated to the application through a callback.
- **Enumeration Complete**:

This event is notified when USB Bus Enumeration is completed and Set Configuration call is received from the USB host. The class driver should now initialize all the endpoints (USB_Device_Init_EndPoint()) except the control endpoint. It should also register callback functions (USB_Device_Register_Service()) to handle endpoint events and set endpoint status as “idle” (USB_Device_Set_Status()).

- **Configuration Change**:

  This event is notified when SET CONFIGURATION call is received from the USB host. Once this event is received, Enumeration Complete event is notified to the class driver.

- **Data Send Complete**:

  This event is notified when data is sent through an endpoint.

- **Data Received**:

  This event is notified when data is received on an endpoint.

Pseudo Code:

```
static void USB_Class_XYZ_Event
(
     uint8_t event,   /* [IN] Event Type*/
     void* val,     /* [IN] additional parameter used by the event */
     void * arg     /* [IN] gives the configuration value */
)
{
     uint8_t index;

     XYZ_DEVICE_STRUCT_PTR  devicePtr;

     devicePtr = (XYZ_DEVICE_STRUCT_PTR)arg;

     if(event == USB_APP_ENUM_COMPLETE)
     {
         uint8_t count = 0;
         uint8_t component;

         /* get the endpoints from the descriptor module */
         USB_ENDPOINTS *ep_desc_data = devicePtr->ep_desc_data;

         /* intialize all non control endpoints */
         while(count < ep_desc_data->count)
         {
             ep_struct_ptr= (USB_EP_STRUCT*) &ep_desc_data->ep[count];
             component = (uint8_t)(ep_struct_ptr->ep_num |
                             (ep_struct_ptr->direction << COMPONENT_PREPARE_SHIFT));
             (void)_usb_device_init_endpoint(devicePtr->handle, ep_struct_ptr,
             TRUE);

             /* register callback service for endpoint */
             (void)_usb_device_register_service(devicePtr->handle,(uint8_t)(USB_SER
             VICE_EP0 + ep_struct_ptr->ep_num), USB_Service_Hid, arg);

             /* set the EndPoint Status as Idle in the device layer */
```

```
            (void)_usb_device_set_status(devicePtr->handle,
            (uint8_t)(USB_STATUS_ENDPOINT|component),(uint16_t)USB_STATUS_IDLE);

            count++;
            }
        }
    else if(event == USB_APP_BUS_RESET)
    {
        <Re-Initialize Class Specific Data Structure >
    }

    if(devicePtr->XYZ_class_callback.callback != NULL)
    {
        devicePtr->XYZ_class_callback.callback(event,val,
        devicePtr->XYZ_class_callback.arg);
    }
}
```

4.2.6 Implement class request routine

This routine is called by the USB Common Class Module. It handles class specific and vendor specific requests received from the USB host. Vendor Specific requests are sent to the Application by using Vendor Specific application callback function already initialized by the class driver.

Pseudo Code:

```
static uint8_t USB_XYZ_Other_Requests
(
    USB_SETUP_STRUCT * setup_packet,/* [IN] setup packet received */
    uint8_t **data,               /* [OUT] data to be send back */
    uint32_t *size,               /* [OUT] size to be returned */
    void * arg
)
{
    uint8_t error = USBERR_INVALID_REQ_TYPE;
    XYZ_STRUCT_PTR obj_ptr = (XYZ_STRUCT_PTR)arg;

    if((setup_packet->request_type & USB_REQUEST_CLASS_MASK) ==
    USB_REQUEST_CLASS_CLASS)
    {
        /* class request so handle it here */
        <Class Specific Code goes here>
    }
    else if((setup_packet->request_type & USB_REQUEST_CLASS_MASK) ==
    USB_REQUEST_CLASS_VENDOR)
    {
        /* vendor specific request  */
        if(obj_ptr->vendor_callback.callback != NULL)
        {
            error = obj_ptr->vendor_callback.callback(setup_packet,data,
            size,obj_ptr->vendor_callback.arg);
        }
    }
```

```
    return error;
}
```

4.2.6.1 Implement endpoint service routine

This routine is called by the USB Low Level Device Framework when data is sent or received on an endpoint. This routine is registered with the Low Level Device Framework by a Class Driver during the endpoint initialization.

Pseudo Code:

```
void USB_Class_XYZ_Service_Endpoint
(
    PTR_USB_EVENT_STRUCT event  /* [IN] pointer to USB Event Structure */
    void * arg /* [IN] gives the configuration value */
)
{
    APP_DATA_STRUCT bulk_data;
    XYZ_STRUCT_PTR obj_ptr = (XYZ_STRUCT_PTR)arg;

    bulk_data.data_ptr = event->buffer_ptr;
    bulk_data.data_size = event->len;
    if(obj_ptr->XYZ_callback.callback != NULL)
    {
        if(event->errors != 0)
        {
            <Class Specific Error Handling Code goes here>
            obj_ptr->XYZ_callback.callback(USB_APP_ERROR, NULL,
            obj_ptr->XYZ_callback.arg);
        }
        else
        {
            if(event->direction == USB_RECV)
            {
                < Class Specific Data Receive Handling Code goes here>
                obj_ptr->XYZ_callback.callback(USB_APP_DATA_RECEIVED,
                    (void*)&bulk_data, obj_ptr->XYZ_callback.arg);
            }
            else
            {
                < Class Specific Data Send Complete Handling Code goes here>
                obj_ptr->XYZ_callback.callback( USB_APP_DATA_SEND_COMPLETE,
                    (void*)&bulk_data, obj_ptr->XYZ_callback.arg);
            }
        }
    }
}
```

Chapter 5 Developing new Applications

5.1 Introduction

This chapter discusses the functions used to develop applications based on the existing classes.

5.2 Application interfaces

The interfaces of the existing classes are defined keeping in mind that the application should be as independent as possible from the lower layer class drivers and USB controller drivers.

The interface definition between the application and classes is made up of the calls shown in the table below.

Table 5-1. API calls

API call	Description
Class Initialize	This API is used to initialize the class that in turn initializes not only the class but also the lower driver layers.
Send Data	This API is used by the application to send the data to the host system. It is not recommended to make this call in a non-interrupt context.
Event Callback	All events on the bus are propagated to the application by using the event callback. The data received on the bus is also propagated to the application by using the event callback.
USB Vendor Specific Callback	This is an optional callback which is not mandatory for the application to support it. This callback is used to propagate any vendor specific request that the host system might have sent.
Periodic Task	This is an API call by the application to the class, so that it can complete some tasks that it may want to execute in non-interrupt context.

5.3 Developing an Application

Perform the following steps to develop a new application:

1. Make a new application directory under /device/examples directory. The new application directory is made to create the new application.
2. Copy the following files from the similar pre-existing applications.
   — usb_descriptor.c
   — usb_descriptor.h

Change these files to suit your application. The usb_descriptor.c and usb_descriptor.h files contain the descriptors for USB that are dependent on the application and the class driver.

3. Create a directory where the project files for the new application can be created.
4. Create a new file for the main application function and the callback function as defined above. In the figure below, the new_app.c and new_app.h are used for the same purpose.

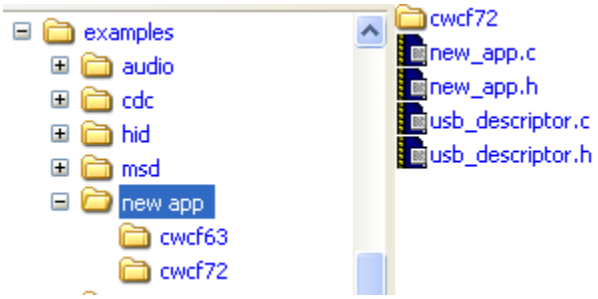

Figure 5-1. New application directory

- usb_descriptor.c

  This file contains USB Framework Module interface. It also contains various descriptors defined by USB Standards such as device descriptor, configuration descriptor, string descriptor, and other class specific descriptors that are provided to the Framework Module when requested. For customization, the user can modify these variables and function implementations to suit the requirement.

  a) Variables

  The list below shows user-modifiable variables for an already implemented class driver. The user should also modify corresponding MACROs defined in the usb_descriptor.h file.

  – **usb_desc_ep**

  This is an array of endpoint structures. Endpoint structure describes the property of the endpoint such as, endpoint number, size, direction, type, and so on. This array should contain all the mandatory endpoints defined by USB class specifications.

  Sample code implementation of the usb_desc_ep for HID class is given below:

```
USB_ENDPOINTS usb_desc_ep =
 {
        HID_DESC_ENDPOINT_COUNT,
        {
                HID_ENDPOINT,
                USB_INTERRUPT_PIPE,
                USB_SEND,
                HID_ENDPOINT_PACKET_SIZE,<---User Modifiable
        }
        < User can add other endpoints depending on class requirement >
 };
```

  – **g_device_descriptor**

  This variable contains USB Device Descriptor.

  Sample code implementation of the device descriptor for HID class is given below:

```
uint8_t const g_device_descriptor[DEVICE_DESCRIPTOR_SIZE] =
{
        DEVICE_DESCRIPTOR_SIZE,         /* "Device Descriptor Size */
        USB_DEVICE_DESCRIPTOR,      /* "Device" Type of descriptor */
        0x00, 0x02,                /* BCD USB version          */
        0x00,                      /* Device Class is indicated in the
```

```
                                        interface descriptors      */
        0x00,                          /* Device Subclass is indicated in
                                        the interface descriptors   */
        0x00,                          /* Device Protocol            */
        CONTROL_MAX_PACKET_SIZE,       /* Max Packet size            */
        0x04,0x25, <---User Modifiable/* Vendor ID                  */
        0x00,0x01, <---User Modifiable/* Product ID                 */
        0x02,0x00,                     /* BCD Device version */
        0x01,      <---User Modifiable/* Manufacturer string index */
        0x02,      <---User Modifiable/* Product string index */
        0x00,      <---User Modifiable /* Serial number string index */
        0x01                              /* Number of configurations */
};
```

– **g_config_descriptor**

This variable contains USB Configuration Descriptor.

Sample code implementation of the configuration descriptor for HID class is given below:

```
uint8_t const g_config_descriptor[CONFIG_DESC_SIZE] =
{
        CONFIG_ONLY_DESC_SIZE,  /*  Configuration Descriptor Size -
                                always 9 bytes */
        USB_CONFIG_DESCRIPTOR,  /* "Configuration" type of descriptor */
        CONFIG_DESC_SIZE, 0x00, /*  Total length of the
                                   Configuration descriptor */
        1,                      /*  NumInterfaces */
        1,                      /*  Configuration Value */
        0,                      /*  Configuration Description String
                                   Index*/
        BUS_POWERED|SELF_POWERED|
        (REMOTE_WAKEUP_SUPPORT<<REMOTE_WAKEUP_SHIFT),
/* CFV1/CFV2 are both self powered (its compulsory to set bus powered)*/
/*Attributes.support RemoteWakeup and self power*/
        0x32,      <---User Modifiable /*  Current draw from bus */

        /* Interface Descriptor */
        IFACE_ONLY_DESC_SIZE,
        USB_IFACE_DESCRIPTOR,
        0x00,
        0x00,
        HID_DESC_ENDPOINT_COUNT,
        0x03,
        0x01,
        0x02,
        0x00,

        /* HID descriptor */
        HID_ONLY_DESC_SIZE,
        USB_HID_DESCRIPTOR,
        0x00,0x01,
        0x00,
        0x01,
        0x22,
```

```
        0x34,0x00,

        /*Endpoint descriptor */
        ENDP_ONLY_DESC_SIZE,
        USB_ENDPOINT_DESCRIPTOR,
        HID_ENDPOINT|(USB_SEND << 7),
        USB_INTERRUPT_PIPE,
        HID_ENDPOINT_PACKET_SIZE, 0x00, <---User Modifiable
        0x0A
};
```

– **String Descriptors**

Users can modify string descriptors to customize their product. String descriptors are written in UNICODE format. An appropriate language identification number is specified in USB_STR_0. Multiple language support can also be added.

Sample code implementation of string descriptors for the HID class application is given below:

```
uint8_t const USB_STR_0[USB_STR_0_SIZE+USB_STR_DESC_SIZE] =
{
        sizeof(USB_STR_0),
        USB_STRING_DESCRIPTOR,
        0x09,
        0x04/*equiavlent to 0x0409*/
        <User can add other language support descriptor here>
};

uint8_t const USB_STR_1[USB_STR_1_SIZE+USB_STR_DESC_SIZE] =
{
        sizeof(USB_STR_1),
        USB_STRING_DESCRIPTOR,
        <User Modifiable Manufacturer Name in UNICODE>
        'F',0,
        'R',0,
        'E',0,
        'E',0,
        'S',0,
        'C',0,
        'A',0,
        'L',0,
        'E',0,
        ' ',0,
        'S',0,
        'E',0,
        'M',0,
        'I',0,
        'C',0,
        'O',0,
        'N',0,
        'D',0,
        'U',0,
        'C',0,
        'T',0,
```

```
        'O',0,
        'R',0,
        ' ',0,
        'I',0,
        'N',0,
        'C',0,
        '.',0
};

uint8_t const USB_STR_2[USB_STR_2_SIZE+USB_STR_DESC_SIZE] =
{
        sizeof(USB_STR_2),
        USB_STRING_DESCRIPTOR,
        <User Modifiable Product Name in UNICODE>
        ' ',0,
        ' ',0,
        'J',0,
        'M',0,
        ' ',0,
        'H',0,
        'I',0,
        'D',0,
        ' ',0,
        'D',0,
        'E',0,
        'M',0,
        'O',0,
        ' ',0
};

uint8_t const USB_STR_n[USB_STR_n_SIZE+USB_STR_DESC_SIZE] =
{
        sizeof(USB_STR_n),
        USB_STRING_DESCRIPTOR,
        'B',0,
        'A',0,
        'D',0,
        ' ',0,
        'S',0,
        'T',0,
        'R',0,
        'I',0,
        'N',0,
        'G',0,
        ' ',0,
        'I',0,
        'N',0,
        'D',0,
        'E',0,
        'X',0
};

uint8_t const g_string_desc_size[USB_MAX_STRING_DESCRIPTORS+1] =
```

```
{
        sizeof(USB_STR_0),
        sizeof(USB_STR_1),
        sizeof(USB_STR_2),
        <User can add other string descriptors sizes here>
        sizeof(USB_STR_n)
};

uint8_t *const g_string_descriptors[USB_MAX_STRING_DESCRIPTORS+1] =
{
        (uint8_t * const) USB_STR_0,
        (uint8_t * const) USB_STR_1,
        (uint8_t * const) USB_STR_2,
        <User can add other string descriptors here>
        (uint8_t * const) USB_STR_n
};

USB_ALL_LANGUAGES g_languages =
{
        USB_STR_0, sizeof(USB_STR_0),
        {
             (uint16_t const)0x0409,
             (const uint8_t **)g_string_descriptors,
             g_string_desc_size
        }
        <User can add other language string descriptors here>
};
```

– **Standard Descriptor Table**

Users can modify standard descriptor table to support additional class specific descriptors and vendor specific descriptors.

Sample implementation below is shown for the HID Class application.

```
uint32_t const g_std_desc_size[USB_MAX_STD_DESCRIPTORS+1] =
{
        0,
        DEVICE_DESCRIPTOR_SIZE,
        CONFIG_DESC_SIZE,
        0, /* string */
        0, /* Interface */
        0, /* Endpoint */
        0, /* Device Qualifier */
        0, /* other speed config */
        <Other Descriptor Sizes goes here>
        REPORT_DESC_SIZE
};

uint8_t * const g_std_descriptors[USB_MAX_STD_DESCRIPTORS+1] =
{
        NULL,
        (uint8_t *)g_device_descriptor,
        (uint8_t *)g_config_descriptor,
```

```
            NULL, /* string */
            NULL, /* Interface */
            NULL, /* Endpoint */
            NULL, /* Device Qualifier */
            NULL, /* other speed config*/
            <Other Descriptor pointers go here>
             Sample HID Class Report Desc->
            (uint8_t *)g_report_descriptor
    };
```

– **g_valid_config_values**

This variable contains valid configurations for a device. This value remains fixed for the device.

```
uint8_t const g_valid_config_values[USB_MAX_CONFIG_SUPPORTED+1]={0,1};
```

– **g_alternate_interface**

This variable contains valid alternate interfaces for a given configuration. Sample implementation uses a single configuration. If user implements additional alternate interfaces, the USB_MAX_SUPPORTED_INTERFACES macro (usb_descriptor.h) should be changed accordingly.

```
static uint8_t g_alternate_interface[USB_MAX_SUPPORTED_INTERFACES];
```

– **desc_callback**

This variable contains all descriptor callback function pointers.

```
DESC_CALLBACK_FUNCTIONS_STRUCT  desc_callback =
{
   0,
   USB_Desc_Get_Descriptor,
   USB_Desc_Get_Endpoints,
   USB_Desc_Get_Interface,
   USB_Desc_Set_Interface,
   USB_Desc_Valid_Configation,
   USB_Desc_Remote_Wakeup
};
```

b) Interfaces

The following interfaces are required to be implemented by the application in the usb_descriptor.c. These interfaces are called by the low level USB stack and class drivers. See the *Freescale MQX™ RTOS USB Device API Reference Manual* (MQXUSBDEVAPI) for details regarding interface functions and sample implementation. Additionally, see the usb_descriptor.c and usb_descriptor.h files in the device\app\hid for reference.

– **USB_Desc_Get_Descriptor**

This interface function is invoked by the USB Framework. This call is made when Framework receives GET_DESCRIPTOR call from the Host. Mandatory descriptors that an application is required to implement are:

– Device Descriptor

– Configuration Descriptor

– Class Specific Descriptors; For example, for HID class implementation, Report Descriptor and HID Descriptor.

Apart from the mandatory descriptors, an application should also implement various string descriptors as specified by the Device Descriptor and other configuration descriptors.

Sample code for the HID class application is given below.

```
uint8_t USB_Desc_Get_Descriptor
(
    HID_HANDLE handle,
    uint8_t type,
    uint8_t str_num,
    uint16_t index,
    uint8_t **descriptor,
    uint32_t *size
)
{
    UNUSED_ARGUMENT (handle)

    switch(type)
    {
        case USB_REPORT_DESCRIPTOR:
            {
                type = USB_MAX_STD_DESCRIPTORS;
                *descriptor = (uint8_t *)g_std_descriptors [type];
                *size = g_std_desc_size[type];
            }
       break;
        case USB_HID_DESCRIPTOR:
            {
                type = USB_CONFIG_DESCRIPTOR ;
                *descriptor = (uint8_t *)(g_std_descriptors [type]+
                    CONFIG_ONLY_DESC_SIZE+IFACE_ONLY_DESC_SIZE);
                *size = HID_ONLY_DESC_SIZE;
            }
        break;
        case USB_STRING_DESCRIPTOR:
            {
                if(index == 0)
                {
                  /* return the string and size of all languages */
                    *descriptor =
                     (uint8_t *)g_languages.languages_supported_string;
                 *size = g_languages.languages_supported_size;
                }
                else
                {
                    uint8_t lang_id=0;
                    uint8_t lang_index=USB_MAX_LANGUAGES_SUPPORTED;

                  for(;lang_id< USB_MAX_LANGUAGES_SUPPORTED;lang_id++)
                    {
                   /* check whether we have a string for this language */
                        if(index ==
                        g_languages.usb_language[lang_id].language_id)
                        {   /* check for max descriptors */
```

```
                                if(str_num < USB_MAX_STRING_DESCRIPTORS)
                              {   /* setup index for the string to be returned */
                                    lang_index=str_num;
                                }
                                break;
                             }
                        }
                        /* set return val for descriptor and size */
                        *descriptor = (uint8_t *)

    g_languages.usb_language[lang_id].lang_desc[lang_index];
                        *size =
                            g_languages.usb_language[lang_id].
                                lang_desc_size[lang_index];
                    }
                }
            break;
             default :
                if (type < USB_MAX_STD_DESCRIPTORS)
                {
                    /* set return val for descriptor and size*/
                    *descriptor = (uint8_t *)g_std_descriptors [type];

                    /* if there is no descriptor then return error */
                    *size = g_std_desc_size[type];

                    if(*descriptor == NULL)
                    {
                        return USBERR_INVALID_REQ_TYPE;
                    }

                }
                else /* invalid descriptor */
                {
                    return USBERR_INVALID_REQ_TYPE;
                }
             break;
        }/* End Switch */
        return USB_OK;
    }
```

– **USB_Desc_Get_Endpoints**

This interface function is called from the class driver. This function returns a pointer to the USB_ENDPOINT structure. This structure describes the characteristics of Non Control Endpoint such as endpoint number, type, direction, and size, or any other endpoint characteristic required by the class driver implementation.

Sample implementation is given below.

```
USB_ENDPOINTS *USB_Desc_Get_Endpoints(HID_HANDLE handle)
{
    UNUSED_ARGUMENT (handle)
    return &usb_desc_ep;
```

```
}
```

– **USB_Desc_Get_Interface**

This interface function is invoked by the USB Framework. The function returns a pointer to alternate interface for the specified interface. This routine is called when USB Framework receives GET_INTERFACE request from the Host.

Sample code for the single configuration HID class is given below.

```
uint8_t USB_Desc_Get_Interface
(
    HID_HANDLE handle,
    uint8_t interface,
    uint8_t *alt_interface
)
{
    UNUSED_ARGUMENT (handle)
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        <User can modify this to support multiple configurations>
        /* get alternate interface*/
        *alt_interface = g_alternate_interface[interface];
        return USB_OK;
    }
    return USBERR_INVALID_REQ_TYPE;
}
```

– **USB_Desc_Remote_Wakeup**

This interface function is invoked by the USB Framework. If the application supports remote wake-up, this function returns TRUE. Otherwise, it returns FALSE. If the application supports remote wake-up, the USB Device Descriptor should support this capability. If the user does not support remote wake-up, the REMOTE_WAKEUP_SUPPORT should be set to 0 in the usb_descriptor.h.

Sample code is given below.

```
bool USB_Desc_Remote_Wakeup(HID_HANDLE handle)
{
    UNUSED_ARGUMENT (handle)
    return REMOTE_WAKEUP_SUPPORT;
}
```

– **USB_Desc_Set_Interface**

This interface function is called from the USB Framework. This function sets an alternate interface for the specified interface. This routine is called when the USB Framework receives SET_INTERFACE request from the Host.

Sample code for single configuration HID class is given below.

```
uint8_t USB_Desc_Set_Interface
(
    HID_HANDLE handle,
    uint8_t interface,
    uint8_t alt_interface
```

```
)
{
    UNUSED_ARGUMENT (handle)
    /* if interface valid */
    if(interface < USB_MAX_SUPPORTED_INTERFACES)
    {
        <User can modify this to support multiple configurations>
        /* set alternate interface*/
        g_alternate_interface[interface]=alt_interface;
        return USB_OK;
    }
    return USBERR_INVALID_REQ_TYPE;
}
```

– **USB_Desc_Valid_Configation**

This interface function is called from the USB Framework. This function returns regardless whether the configuration is valid or not. This routine is called when USB Framework receives SET_CONFIGURATION request from the Host.

Sample code for a single configuration HID class is given below.

```
bool USB_Desc_Valid_Configation
(
    HID_HANDLE handle,
    uint16_t config_val
)
{
    uint8_t loop_index=0;
    UNUSED_ARGUMENT (handle)

    /* check with only supported val right now */
    while(loop_index < (USB_MAX_CONFIG_SUPPORTED+1))
    {
        if(config_val == g_valid_config_values[loop_index])
        {
            return TRUE;
        }
        loop_index++;
    }
    return FALSE;
}
```

Apart from the above interfaces mandated by USB Low Level Framework, the application must also implement various callback functions to receive events from the USB class driver. These interface functions are implemented in the new_app.c file.

- new_app.c

— **Class Callback function**

This function is used by a class driver to inform an application about various USB Bus Events. The application can use these events to perform event specific functionality. The implementation of this callback function is governed by the class driver specification.

Pseudo Code:

```
void USB_App_Callback(
      uint8_t controller_ID,/* [IN] Controller ID */
      uint8_t event_type,   /* [IN] value of the event*/
      void* val            /* [IN] gives the configuration value*/
)
{
    UNUSED (controller_ID)
    UNUSED (val)
    if(event_type == USB_APP_BUS_RESET)
    {
        /* USB Bus Reset */
        <Application Specific Code goes here>
    }
    else if(event_type == USB_APP_ENUM_COMPLETE)
    {
        /* Enumeration is complete */
        <Application Specific Code goes here>
    }
    return;
}
```

— **Vendor Request Callback**

This optional function allows an application to support any vendor specific USB requests received from the USB host. This function allows the application developer to enhance the existing class implementation by adding a vendor specific functionality.

Pseudo Code:

```
uint8_t USB_App_Vendor_Request_Callback
(
        uint8_t request,      /* [IN] request type */
        USB_SETUP_STRUCT *setup, /* [IN] pointer to Setup Packet */
         uint8_t ** data,    /* [OUT] pointer to the data */
         USB_PACKET_SIZE* size/* [OUT] size of the transfer */
)
{
        uint8_t status = USB_OK;
        uint8_t request = setup-> request;
        switch(request)
        {
             < Vendor Specific Requests are handled here >
            case <VENDOR_REQUEST1> :
                 *data = <pointer to Data to be sent>
                 *size = <size of Data to be sent>
                 break;
            case <VENDOR_REQUEST2>:
                 *data = <pointer to Data to be sent>
                 *size = <size of Data to be sent>
                 break;
                      :
                      :
                      :
            default:
                       < UNHANDLED Vendor Specific Requests
```

```
                        Application code goes here>
                        status = USBERR_INVALID_REQ_TYPE;
                        break;
                }
                return status;
        }
```

— **Class Specific Request Callback**

Implementation of this function is governed by the Class Driver Design. The design and implementation details are, however, beyond the scope of this document.

Pseudo Code:

```
uint8_t USB_App_Class_Spec_Request_Callback
(
    uint8_t request,      /* [IN] request type */
    uint16_t value,       /* [IN] report type and ID */
    uint8_t ** data,    /* [OUT] pointer to the data */
    USB_PACKET_SIZE* size/* [OUT] size of the transfer */
)
{
    uint8_t status = USB_OK;
    *size = 0;
    /* handle the class request */
    switch(request)
    {
        < Class Specific Requests are handled here >
        case <REQUEST1>:
        *data = <pointer to Data to be sent>
        *size = <size of Data to be sent>
        break;
        case <REQUEST2>:
        *data = <pointer to Data to be sent>
        *size = <size of Data to be sent>
        break;
        :
        :
        :
        default:
        < UNHANDLED Class Specific Requests Application code goes here>
        status = USBERR_INVALID_REQ_TYPE;
        break;
    }
return status;
}
```

- usb_descriptor.h

  This file is mandatory for the application. The framework and class drivers include this file for function prototype definitions and data structures described in usb_descriptor.c. The user modifying the usb_descriptor.c should also modify the MACROs in this file.

5. After the functionality has been implemented, use the steps defined in Section A.1.2, “Building the Application with CodeWarrior 6 and CodeWarrior 7 and Section A.1.3, “Running the Application with CodeWarrior 6 and CodeWarrior 7 to build and run the application.

5.4 Application design

This section discusses the application design. The application consists of the main application function and the callback function.

5.4.1 Main Application Function

The main application function uses the following C code:

```
void TestApp_Init(void)
{
    HID_CONFIG_STRUCT   config_struct;

    _int_disable();
    <Application Specific Initialization Code goes here>
    /* initialize the Global Variable Structure */
    USB_mem_zero(&g_mouse, sizeof(MOUSE_GLOBAL_VARIABLE_STRUCT));
    USB_mem_zero(&config_struct, sizeof(HID_CONFIG_STRUCT));

    /* Initialize the USB interface */

    config_struct.ep_desc_data = &usb_desc_ep;
    config_struct.hid_class_callback.callback =USB_App_Callback;
    config_struct.hid_class_callback.arg = &g_mouse.app_handle;
    config_struct.param_callback.callback =USB_App_Param_Callback;
    config_struct.param_callback.arg = &g_mouse.app_handle;
    config_struct.desc_callback_ptr = &desc_callback;
    config_struct.desc_endpoint_cnt = HID_DESC_ENDPOINT_COUNT;
    config_struct.ep = g_mouse.ep;

     <USB Class Initialization Call>
    g_mouse.app_handle = USB_Class_HID_Init(&config_struct);
    _int_enable();
    while (TRUE)
    {
         <Application Specific Code goes here>
        /* call the periodic task function */
        USB_HID_Periodic_Task();
    }
}
```

5.5 USB DDK Changes in MQX RTOS version 3.8.1

The USB Host and Device stack were significantly modified in the Freescale MQX RTOS version 3.8.1. The aim was to support both the device and the host in one application. These changes led to the modifications of both internal and external API.

The common part of USB stacks was moved into *usb/common* folder. In this folder you can find common header files and example application demonstrating both host and device functionality at the same time.

The public common headers are copied after USB-HDK or USB-DDK build directly into *lib/<board>.<comp>/usb* folder.

An application developer should be aware of following changes if porting from the previous MQX versions:

BSP:

- Added new file (*init_usb.c*) containing initialization structures for USB controllers available on board.

USB-DDK:

- `USB_STATUS _usb_device_driver_install(usb_host_if_struct *)` function added.
- `USB_STATUS _usb_device_init(usb_device_if_struct *)` API changed.

The usage of the install and init functions is as simple as possible. In the <board>.h file you can find exported default USB host controller interface and default USB device controller. For example:

```
#define USBCFG_DEFAULT_DEVICE_CONTROLLER  (&_bsp_usb_dev_khci0_if)
```

You can use these macros in the `_usb_device_driver_install` and `_usb_device_init` functions.

5.5.1 Migration steps for USB device application

1. Application project - add compiler search path (*lib/<board>.<comp>/usb*) to make common header files (*usb.h, ...*) available for the compiler.The path should have priority over the other USB paths.
2. Use the following included in your application.

```
#include <mqx.h>
#include <bsp.h>
#include <usb.h>
#include <devapi.h> /* if you want to use USB-DDK */
/* Additionally, include USB device class driver header files… */
```

3. Install the low level driver:

```
res = _usb_device_driver_install(USBCFG_DEFAULT_DEVICE_CONTROLLER);
```

4. If not initialized in the USB-DDK class driver, initialize the low level driver before calling any class initialization function:

```
res = _usb_device_init(USBCFG_DEFAULT_DEVICE_CONTROLLER, &device_handle,
num_of_endpoints);
```

Appendix A Working with the Software

A.1 Introduction

This chapter gives you an insight into using the Freescale MQX USB Device Stack software. The following sections are described in this chapter:

- Preparing the setup
- Building the application
- Running the application

The HID mouse application for the MCF52259 Demo is used as an example in this chapter. Kinetis, CFV1, and CFV2 devices are used as an example to prepare setup, building, and running the application.

A.1.1 Preparing the setup

A.1.1.1 Hardware setup

- Make the connections as shown in Figure A-1. Shown here is hardware setup for MCF52259 Demo board.

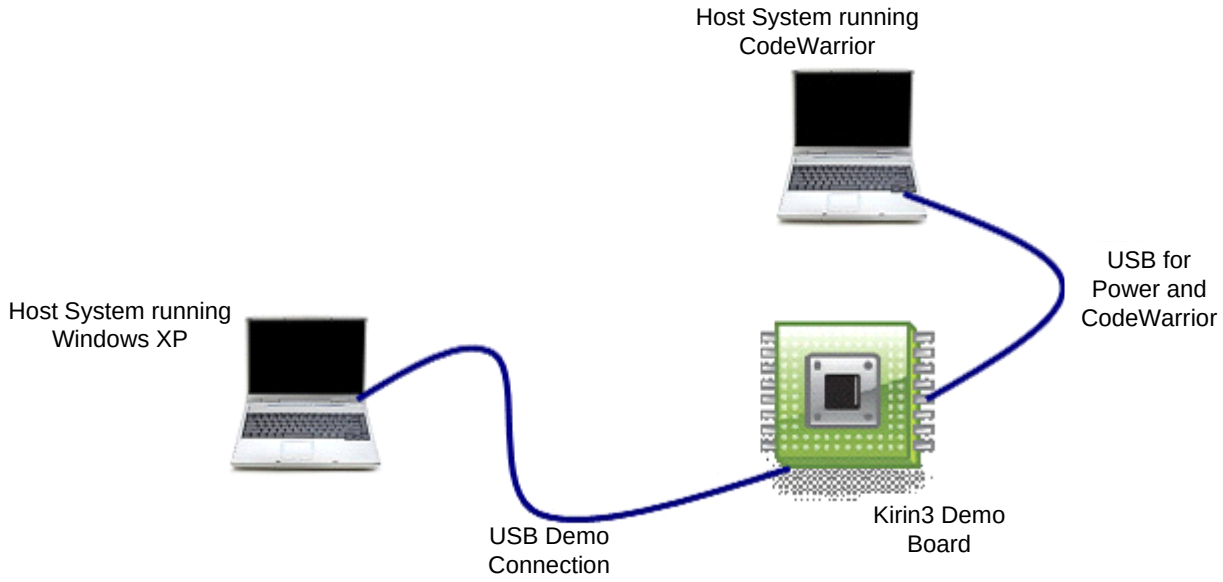

Figure A-1. Hardware setup

— Make the first USB connection between the personal computer, where the software is installed, and the MCF52259 Demo board, where the silicon is mounted. This connection is required to provide power to the board and to download the image to the flash.

— Make the second connection between the MCF52259 Demo board and the personal computer where the demo is run.

NOTE

Although we have used two personal computers in Figure A-1, in reality, you may achieve the same result by a single personal computer with two or more USB ports.

A.1.2 Building the Application with CodeWarrior 6 and CodeWarrior 7

The software for CFV1 is built with CodeWarrior 6.3. and the software for CFV2 is built with CodeWarrior 7.2. Therefore, the software contains application project files that can be used to build the project.

Before starting to build the project, make sure that the CodeWarrior 7.2 is installed on your computer.

To build the CFV2 project:

1. Navigate to the project file and open the hid_mouse_dev_m52259demo.mcp project file in the CodeWarrior IDE.

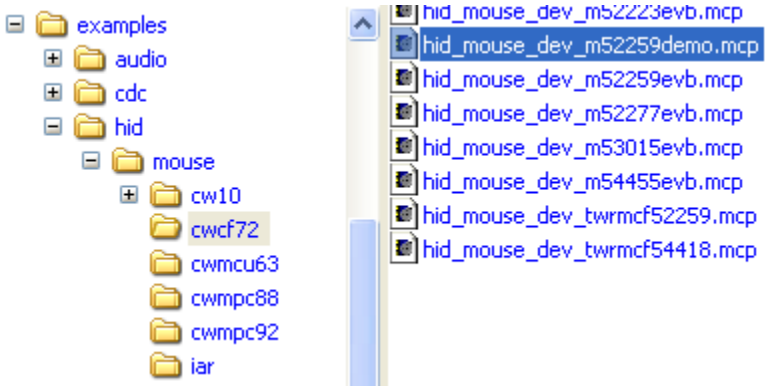

Figure A-2. Open hid_mouse_dev_m59959demo.mcp project file

2. After you have opened the project, the following window appears. To build the project, click the button as shown in Figure A-3.

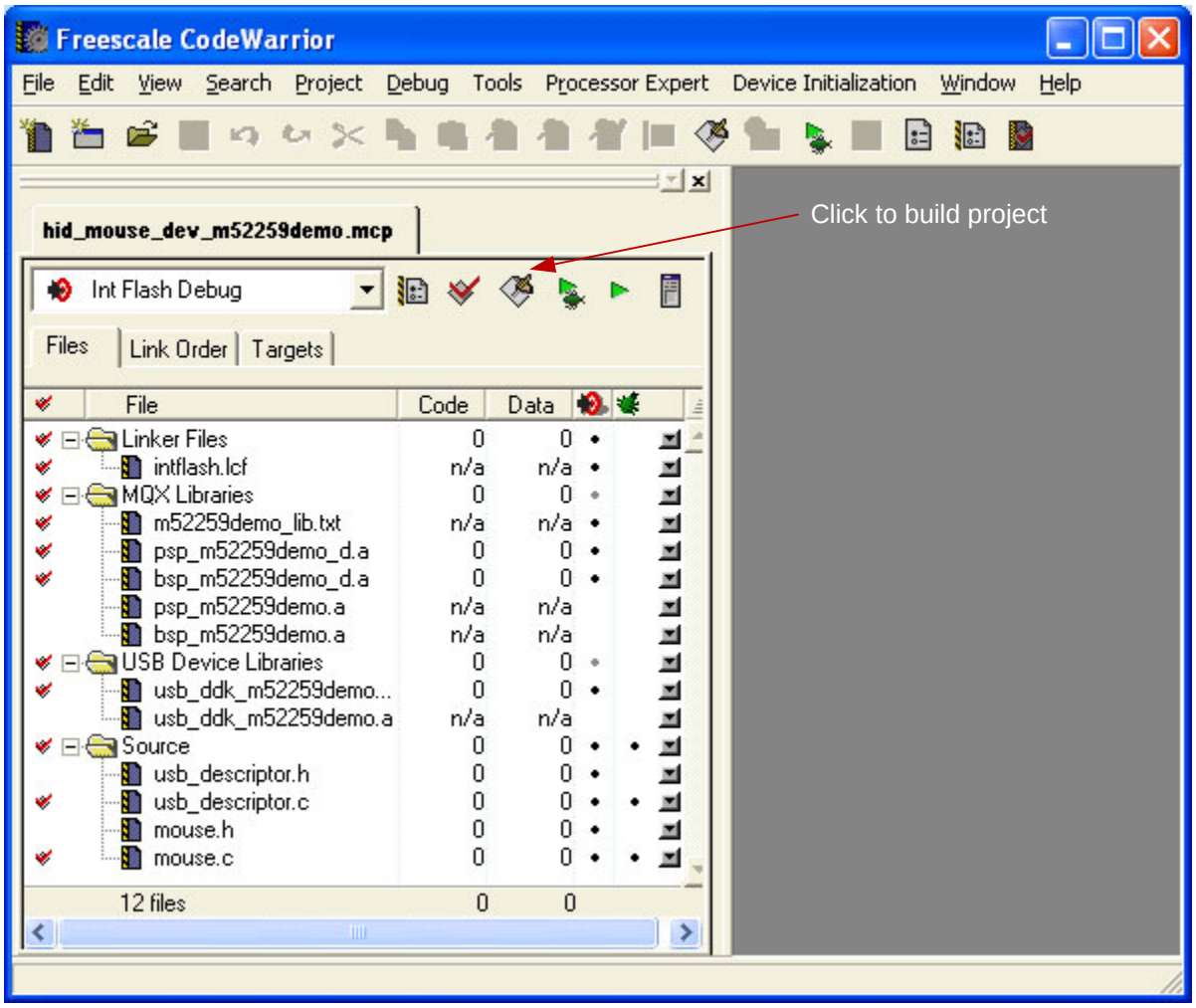

Figure A-3. Build hid_mouse_dev_m52259demo.mcp project

3. After the project is built, the code and data columns must appear filled across the files.

A.1.3 Running the Application with CodeWarrior 6 and CodeWarrior 7

See the board documentation and CodeWarrior Reference Manual for details about programing the flash memory on the evaluation board. The following steps are presented as an example about how to run the HID mouse application with MCF52259 demo board by using an OSBDM debugger.

1. To load the built image to the board, choose Flash Programmer as shown in Figure A-4.

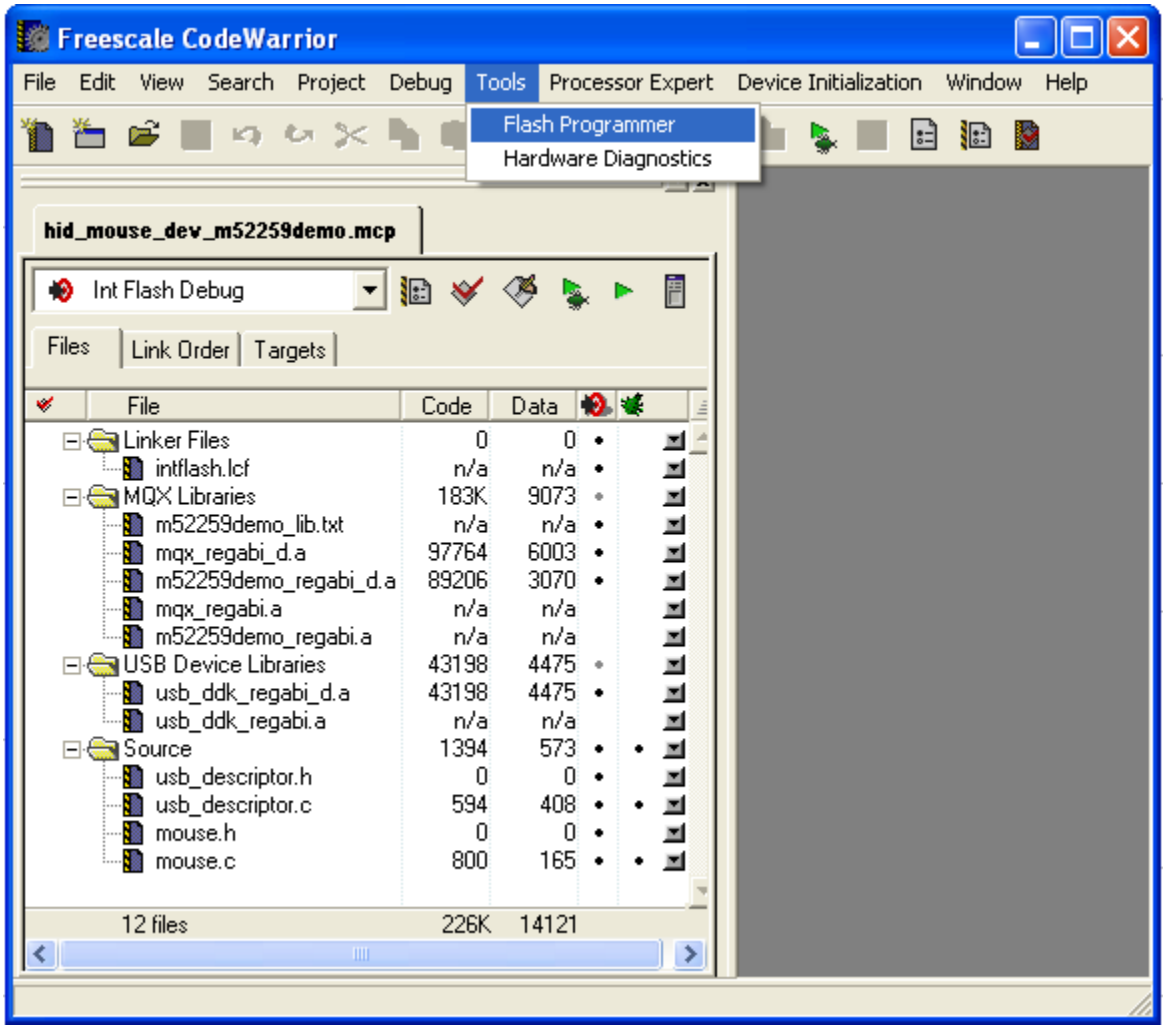

Figure A-4. Open Flash Programmer

2. Click “Load Settings” button and navigate to the memory configuration file of MCF52259 as shown in the following figure.

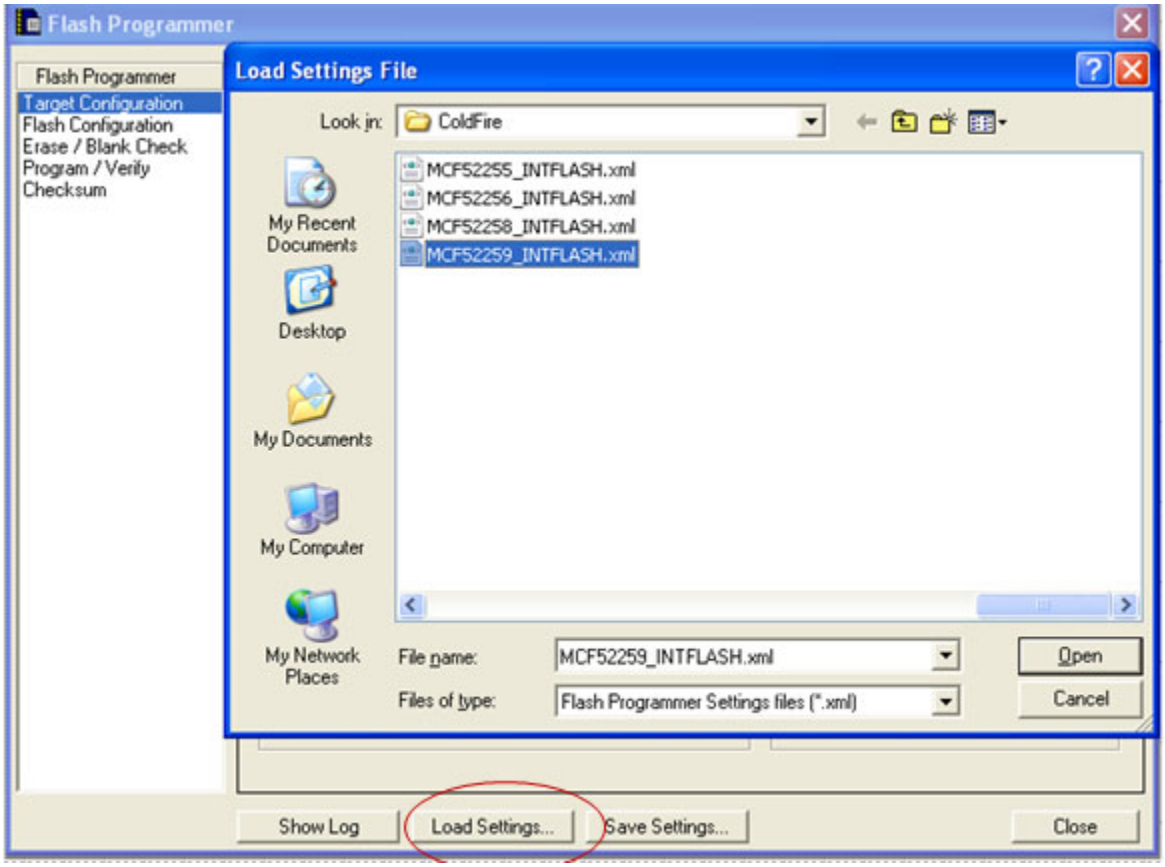

Figure A-5. Select M52259 flash configuration file

3. Click the “Open” button.
4. Choose “Erase/Blank Check” item in the menu and click “Erase” button to erase the old image on the board.

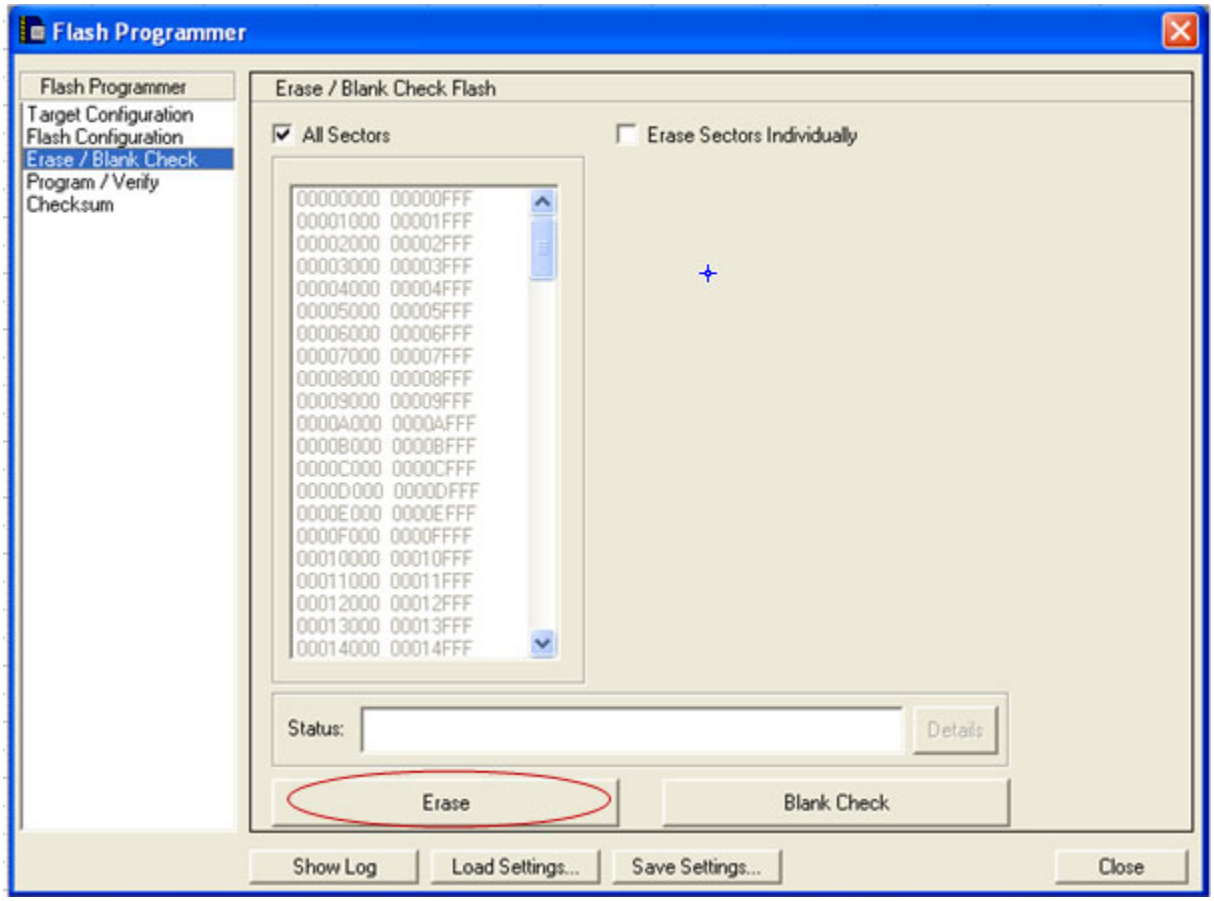

Figure A-6. Erase image in the board

5. Choose “Program/Verify” item in the menu and click “Program” button to load built image to the board.

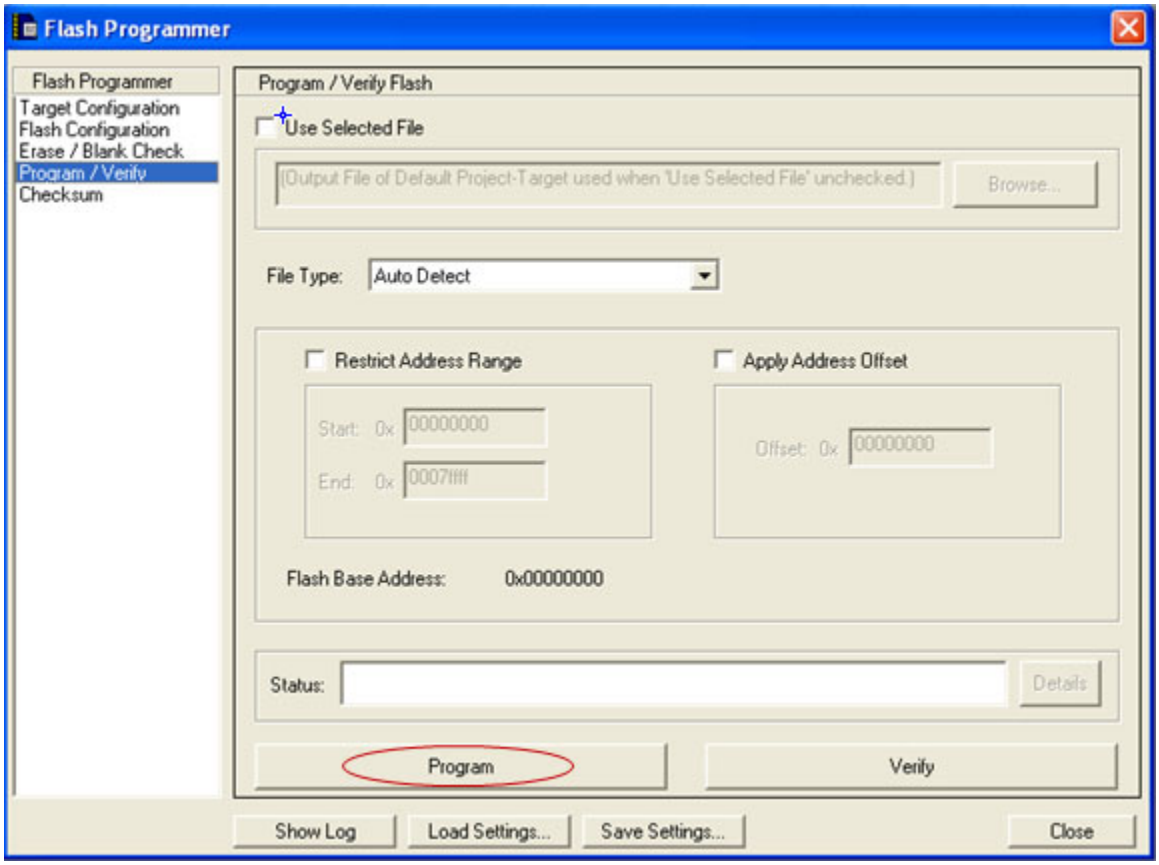

Figure A-7. Program image into the board

6. After the image is programmed in the flash, the debugger window appears as shown in Figure A-8.

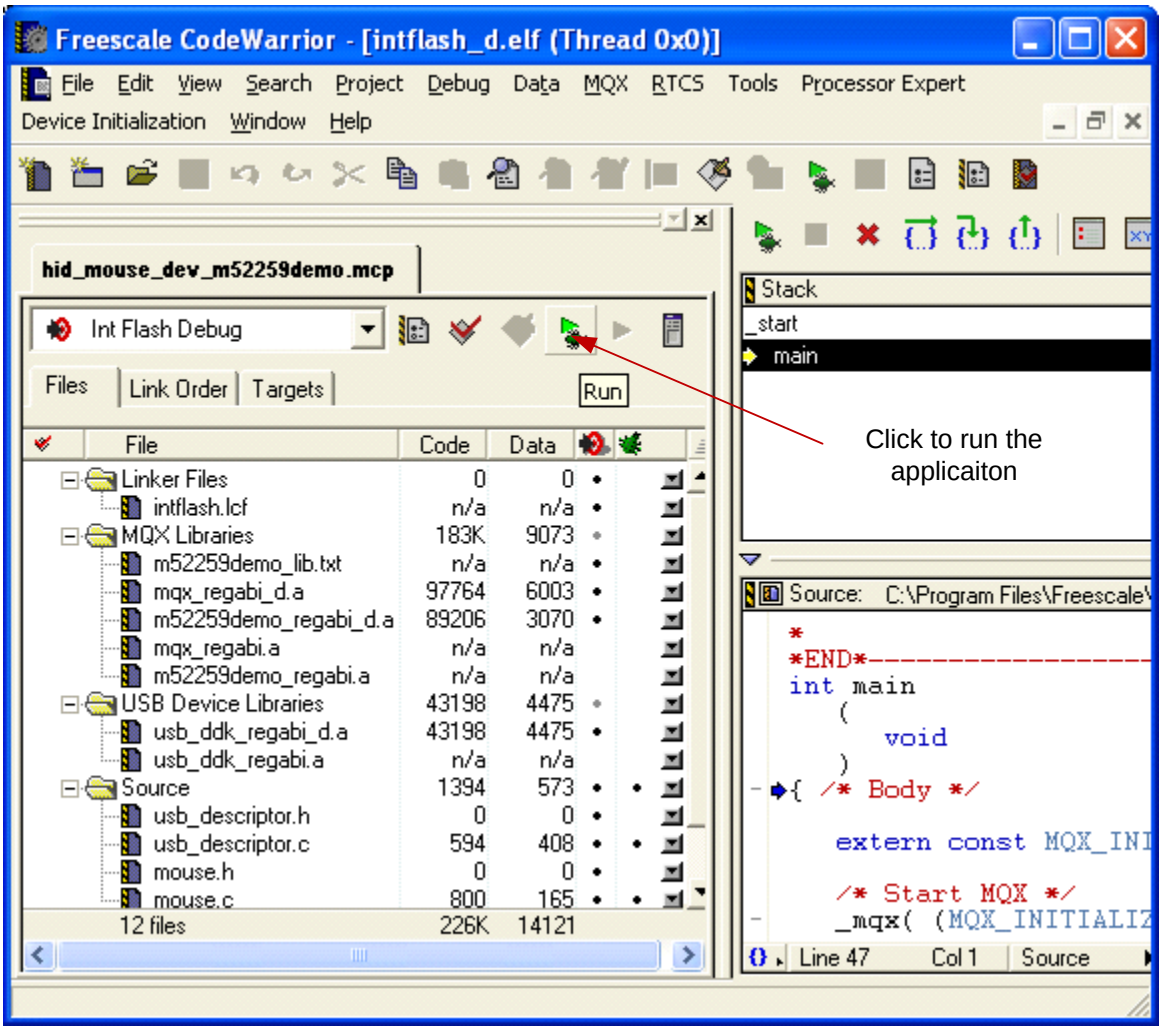

Figure A-8. Debugging the application on CW7.2

A.1.4 Building and Running the Application with CodeWarrior 10

The software for Kinetis k40, CFV1, and CFV2 targets is available to be built, downloaded, and debugged by using the CodeWarrior 10 MCU.

Before starting the the project building process, make sure that the CodeWarrior 10 MCU is installed on your computer.

To build the project (for example Kinetis k40/CFV1/CFV2):

1. Navigate to the project folder (hid_mouse_dev_m52259demo) and locate the CodeWarrior10 project file (.project).

.

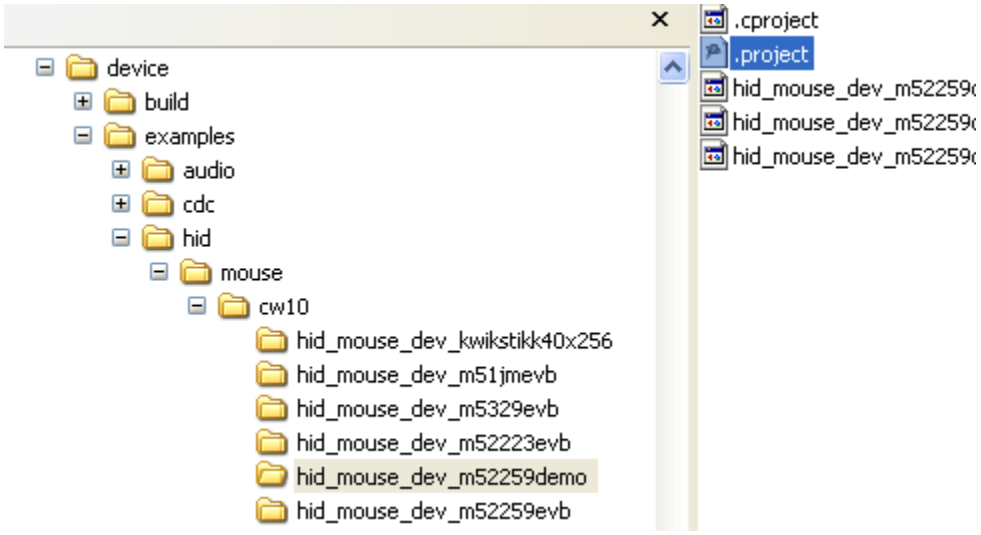

Figure A-9. hid_mouse_dev_m59959demo CW10 project file

2. Open the project by dragging the .project file and dropping it into the CodeWarrior 10 project space.

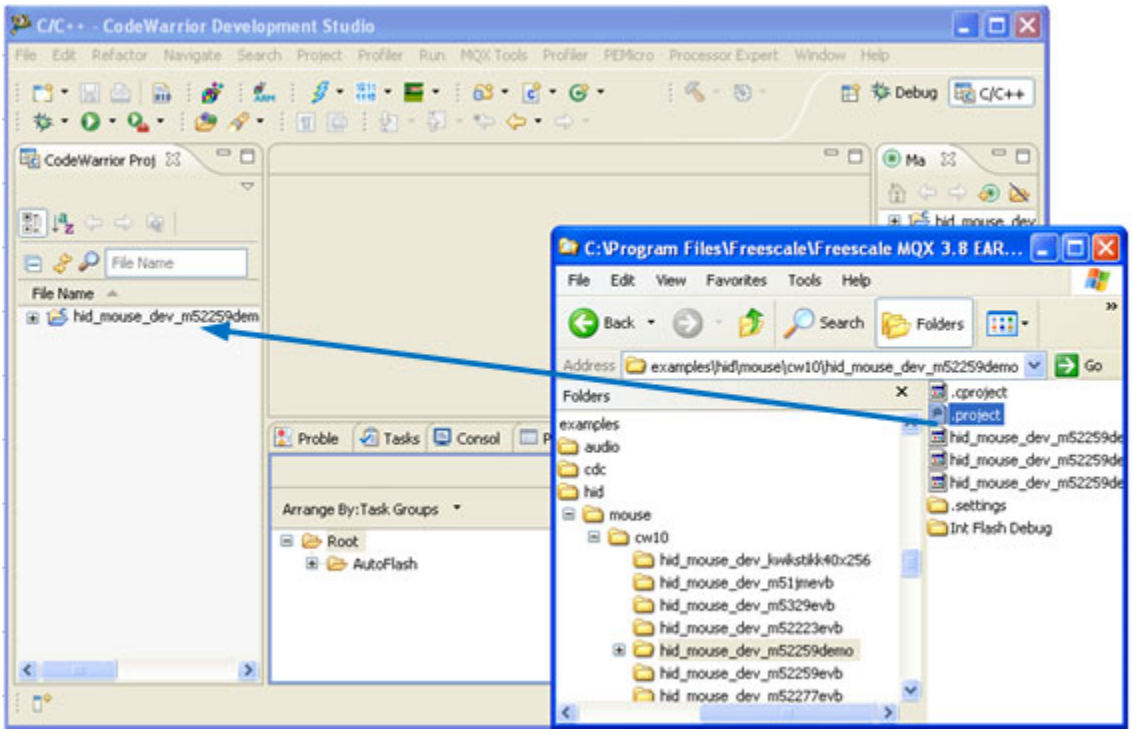

Figure A-10. Open hid_mouse_dev_m59959demo CW10 project file

3. After you have opened the project, the following window appears. To build the project, choose "Build Project" from the Project menu.

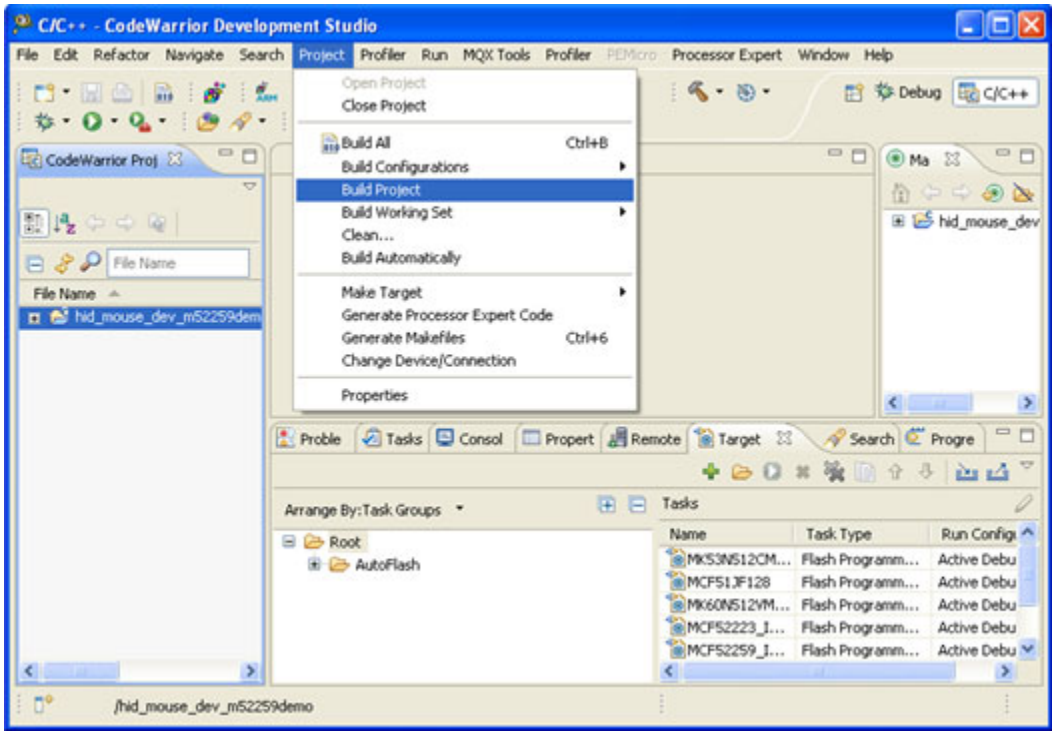

Figure A-11. Build image in CW10

4. To run the application, first locate the hid_mouse_dev_m52259demo_Int_Flash_Debug_OSBDM.launch configuration in the current project space. Right-click it and choose Debug As > 1 hid_mouse_dev_m52259demo_Int_Flash_Debug_OSBDM as shown in the figure below.

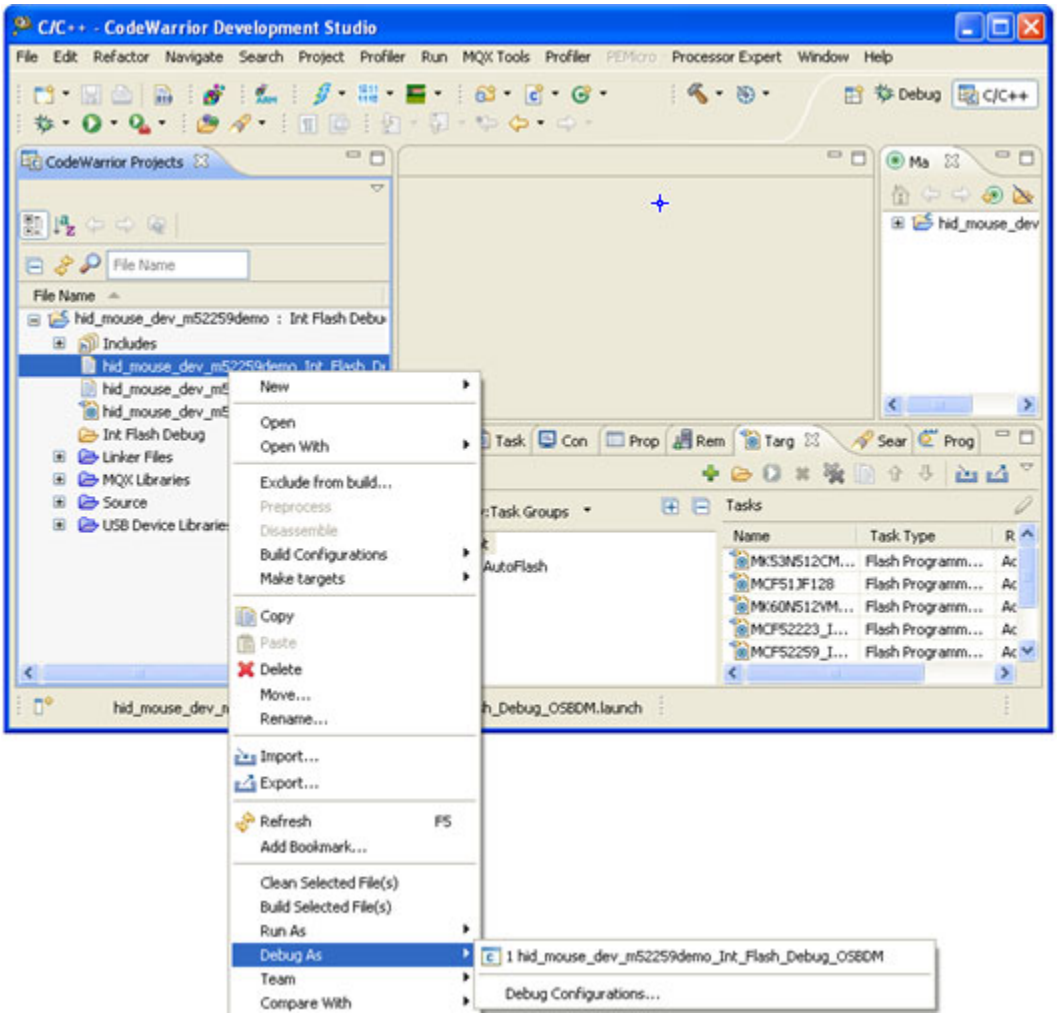

Figure A-12. Choose debug configuration for project

5. After the image is programmed in the flash, the debugger window appears as shown in the following figure. Click on the Green arrow in the Debug tab to run the image.

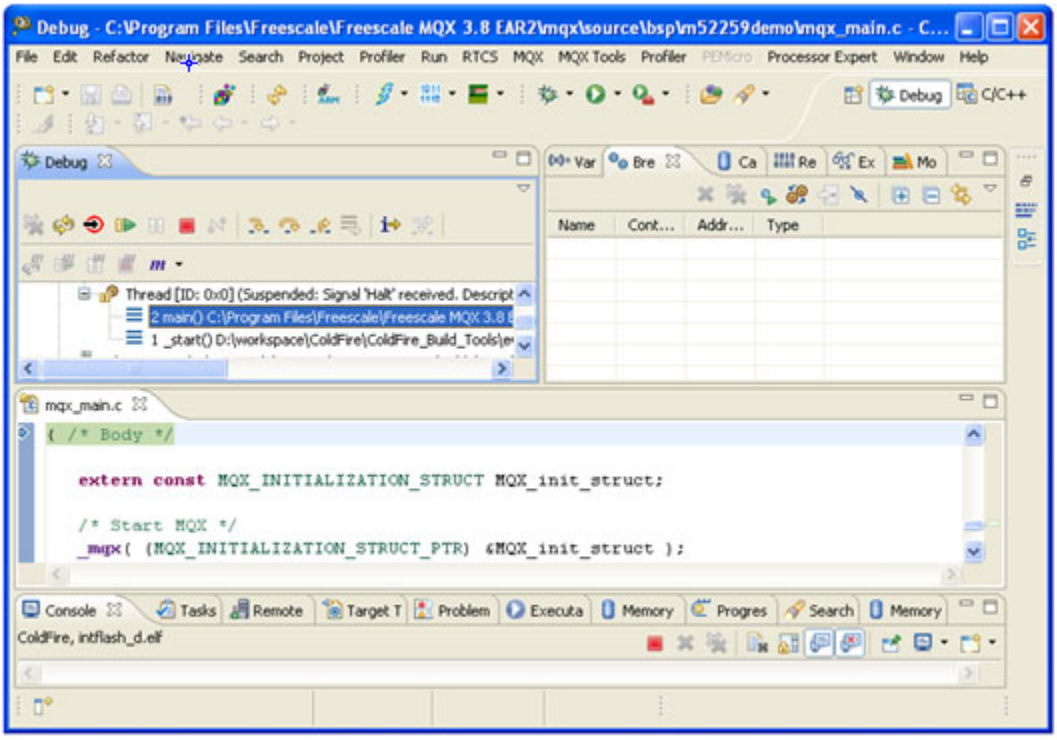

Figure A-13. CW10 debugging window

Appendix B Human Interface Device (HID) Demo

B.1 Setting up the demo

Set the systems as described in the Section A.1.1.1, “Hardware setup.

B.2 Running the demo

After the HID application is programmed into the processor flash memory, the demo can be run using the following procedure.

1. Connect the hardware to the Windows host computer. As soon as you turn on the device, the HID device gets installed onto the Windows host computer. You should see the callout, as shown in Figure B-1, on the right bottom corner of your screen. At this point, the windows installs the host driver for the device.

Figure B-1. Find New Hardware callout

2. To verify whether the mouse has been properly installed or not, see the MCF52259 device entry in the device manager.

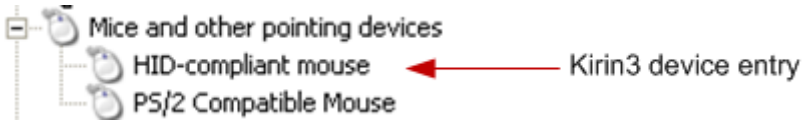

Figure B-2. MCF52259 device entry

3. After the HID device is installed, you will see the mouse moving.

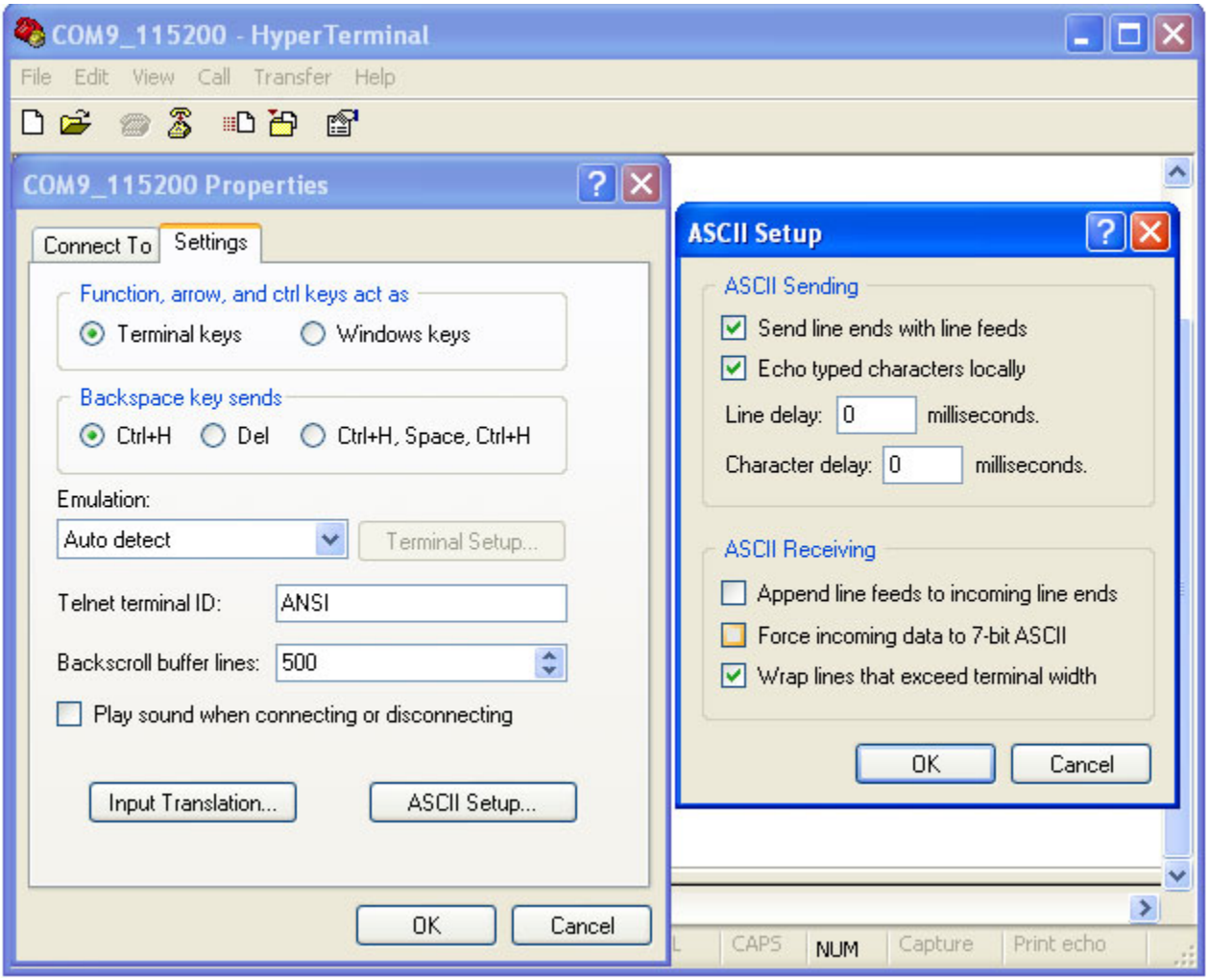

Figure B-3. Configure COM9_115200 - HyperTerminal

4. The HyperTerminal is configured now. Whatever is typed will be echoed back from the virtual COM port. Therefore, it appears in duplicate as shown in Figure B-4.

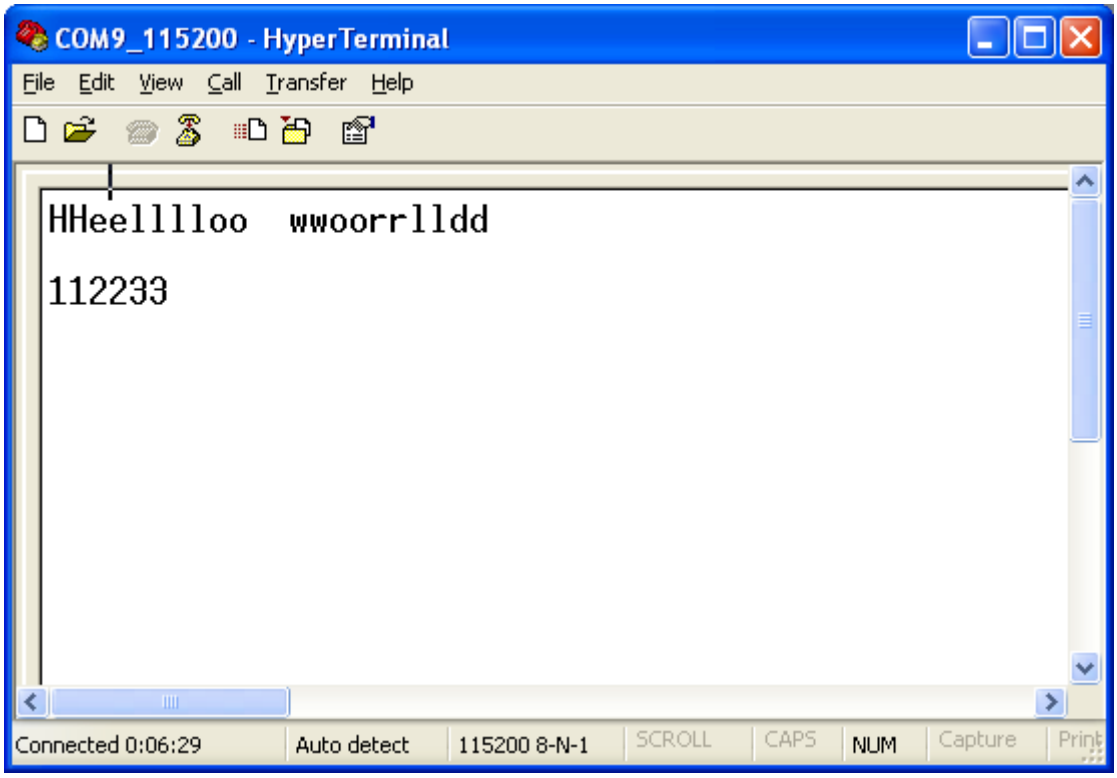

Figure B-4. COM9_115200 - HyperTerminal

Appendix C Virtual Communication (COM) Demo

USB to Serial demo implements the Abstract Control Model (ACM) subclass of the USB CDC class that enables the serial port applications on the host PC to transmit and receive serial data over the USB transport.

C.1 Setting up the demo

Set the systems as described in the Section A.1.1.1, “Hardware setup.

C.2 Running the demo

After the system has been set, follow these steps to run the demo:

1. Turn on the MCF52259 demo board. “Found New Hardware” window appears.

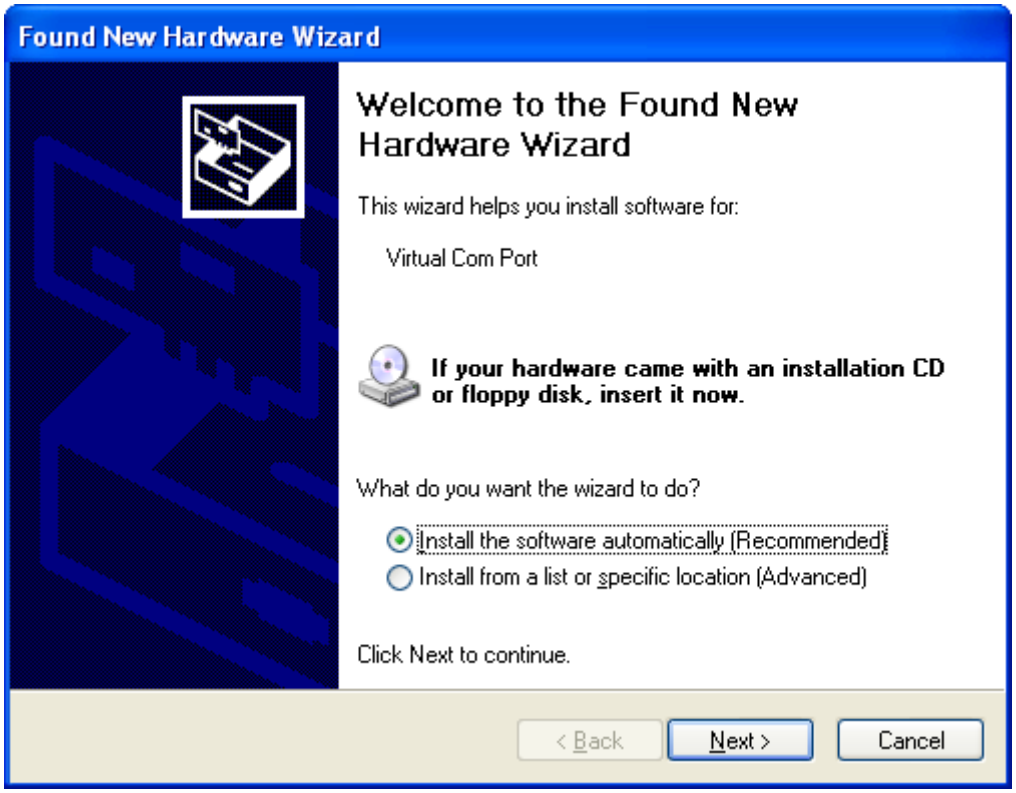

Figure C-1. Found New Hardware window

2. Select “Install from a list or specific location (Advanced)” option as shown in Figure C-1, and click on the “Next” button. “Search and installation options” window appears as shown in Figure C-2.

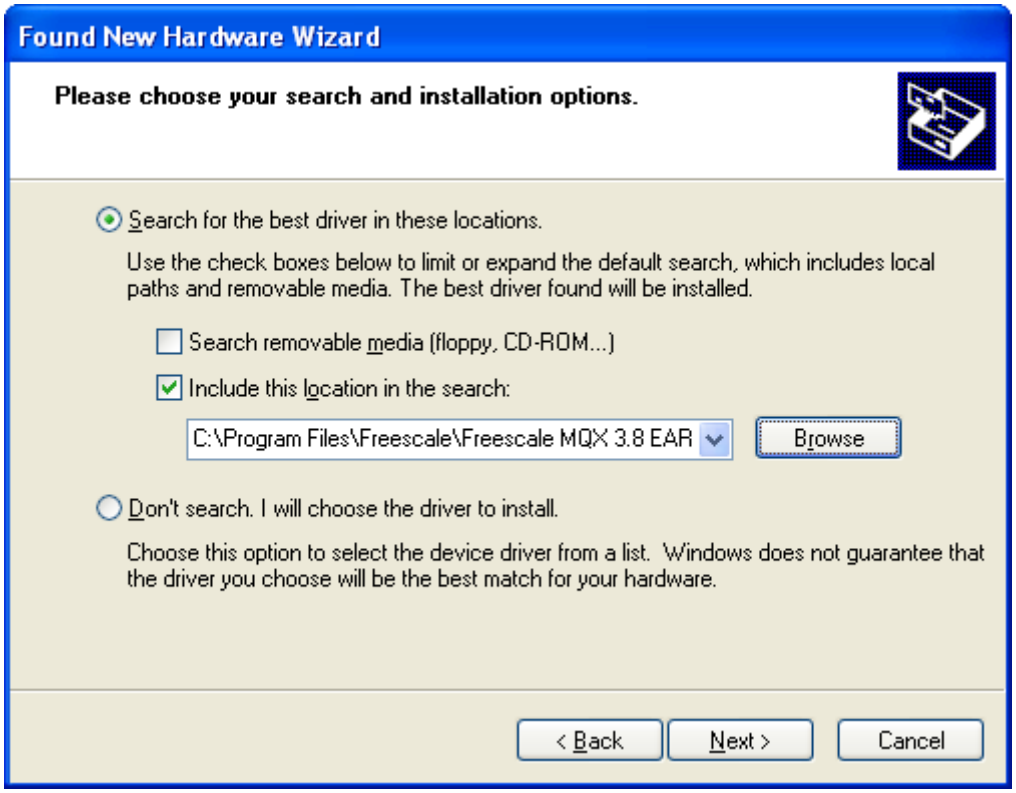

Figure C-2. Search and installation options

Point the search path to the inf directory as shown Figure C-2 and click on the “Next” button. The driver for the device will get installed. To verify the installation, open the device manager. You should see the FSL Virtual COM Port device entries.

Figure C-3. USB CDC device entry in device manager

3. Open the HyperTerminal application as shown in Figure C-4.

Figure C-4. Launch HyperTerminal application

4. The HyperTerminal opens as shown in Figure C-5. Enter the name of the connection and click on the OK button.

Figure C-5. HyperTerminal GUI

5. The window appears as shown in the Figure C-6 . Select the COM port identical to the one that shows up on the device manager.

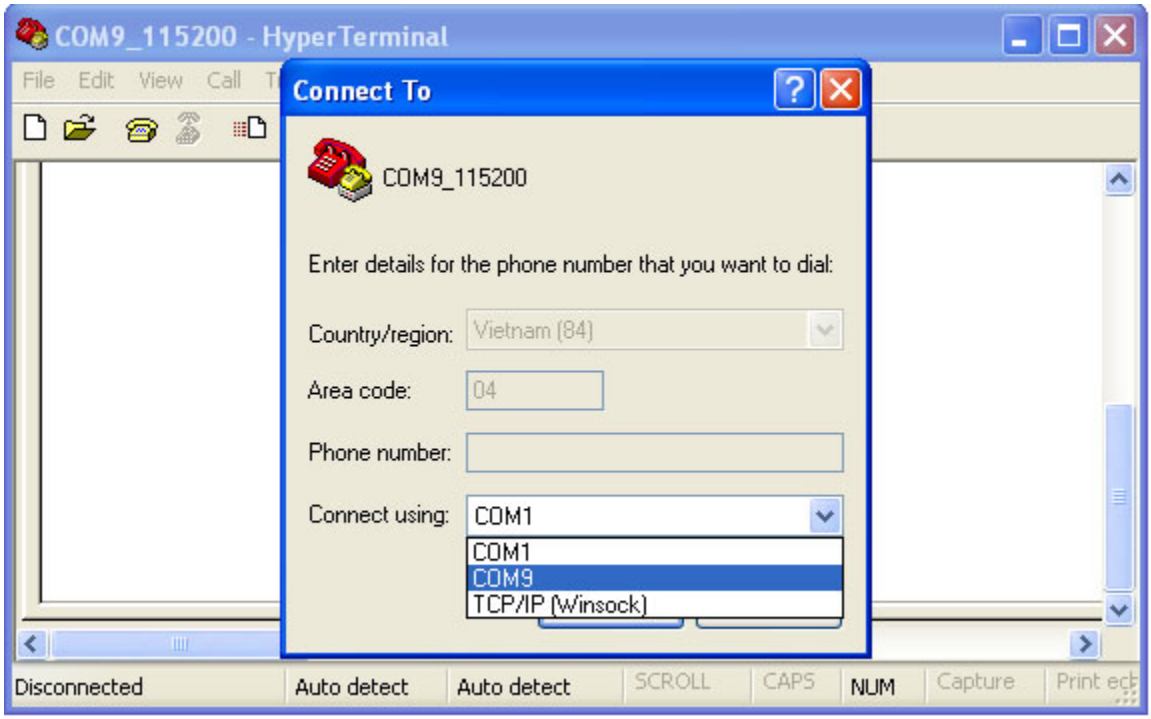

Figure C-6. Connect Using COM9

6. Configure the virtual COM port baud rate and other properties as shown in Figure C-7.

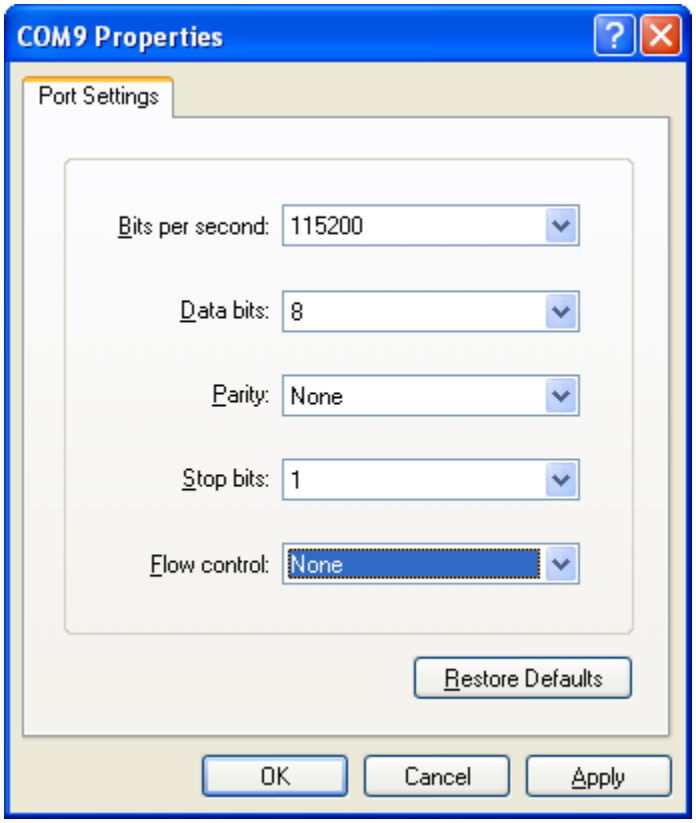

Figure C-7. COM9 properties

7. Configure the HyperTerminal as shown in Figure C-8. Click on the “OK” button to submit changes.

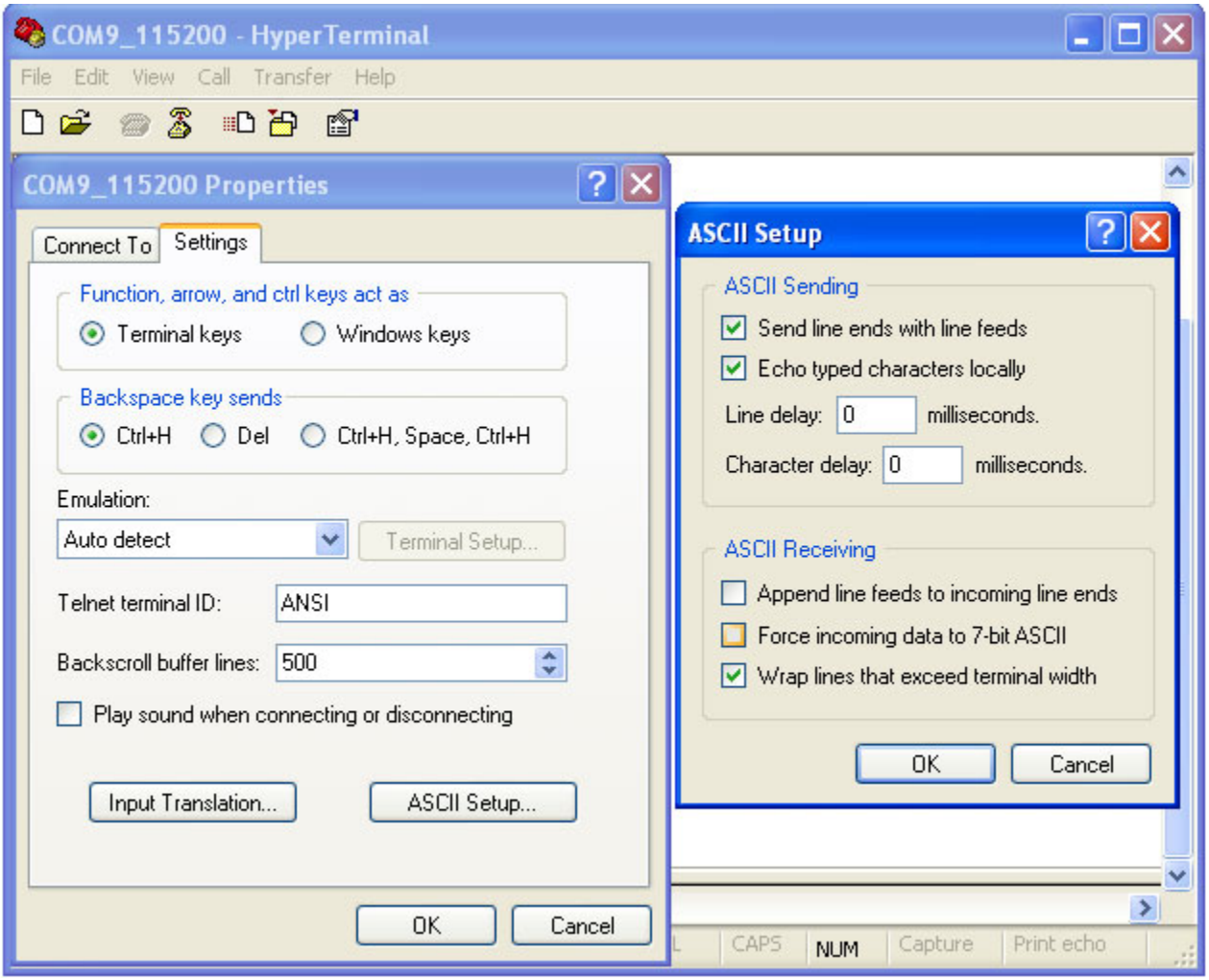

Figure C-8. Configure COM9_115200 - HyperTerminal

8. The HyperTerminal is configured now. Whatever is typed will be echoed back from the virtual COM port. Therefore, it appears in duplicate as shown in Figure C-9.

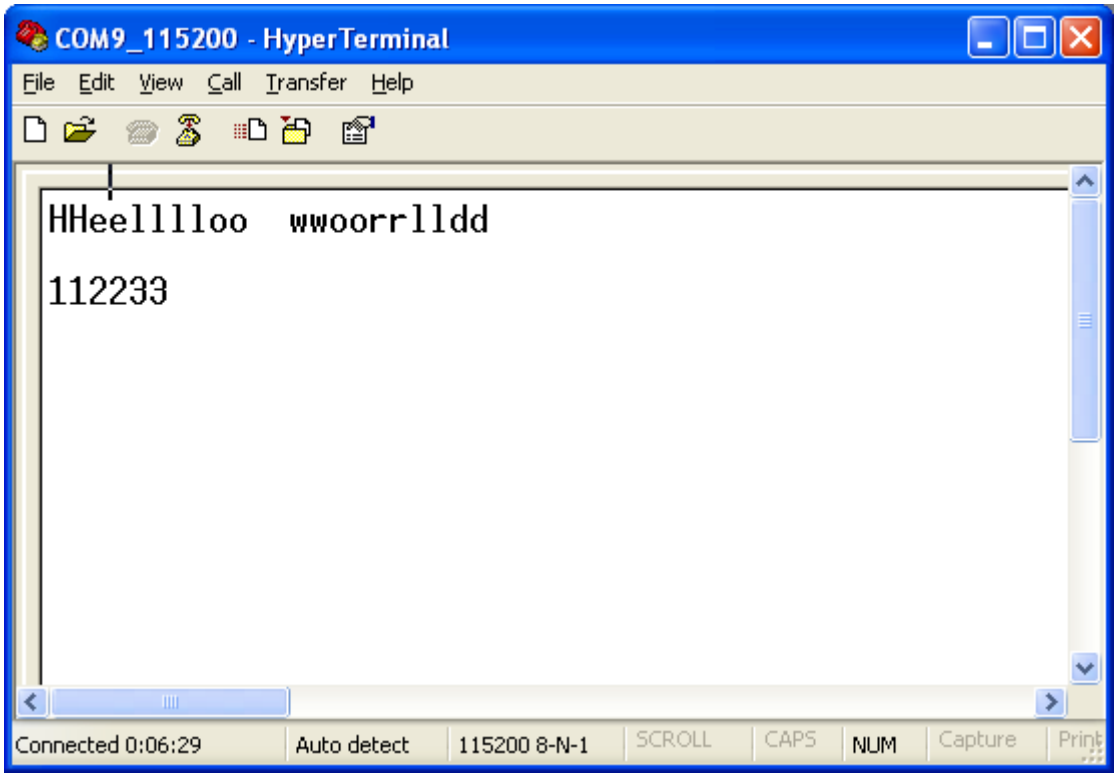

Figure C-9. COM9_115200 - HyperTerminal

Appendix D MSD Demo

The MSD demo implements the SCSI subclass of the USB MSC class. On running the application, MSD device is available as removable disk in Windows.

D.1 Setting up the demo

Set the systems as described in the Section A.1.1.1, “Hardware setup.

D.2 Running the demo

After the MSD application is programmed into the silicon flash, you should follow these steps to run the demo:

1. Connect the hardware to Microsoft Windows host computer. As soon as you turn on the device, the MSD device gets installed onto the host computer. You should be able to see the callout, as shown in Figure D-1, on the right bottom corner of your screen. At this point, Microsoft Windows installs the host driver for the device.

Figure D-1. Found New Hardware callout

2. To verify whether the MSD device has been properly installed, or detected by the Microsoft Windows, you should be able to see the MCF52259 device entry in the device manager.

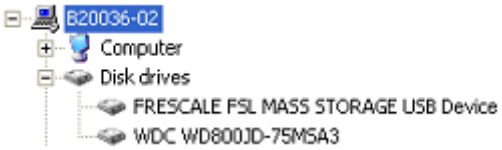

Figure D-2. MCF52259 device entry

3. After the MSD device is installed, you can perform read, write, and format operations on it.

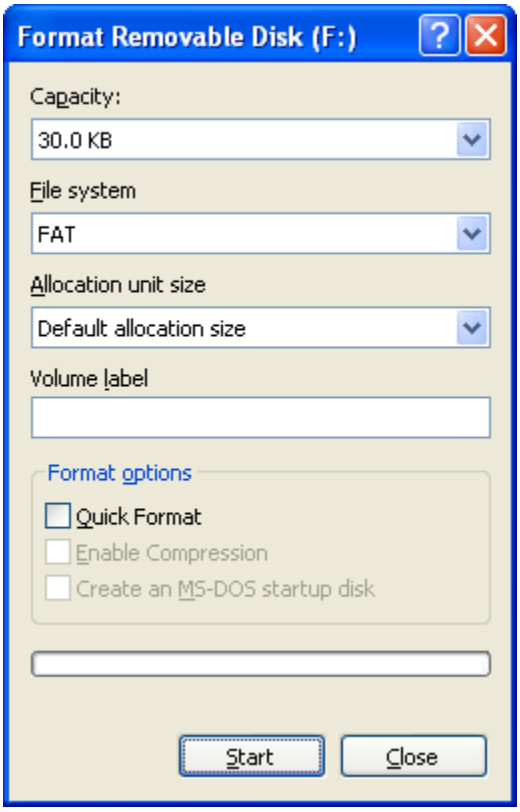

Figure D-3. Option to format the MSD device