Freescale MQX™ RTOS USB Host User's Guide

MQXUSBHOSTUG
Rev. 7
08/2014

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: MQXUSBHOSTUG
Rev. 7, 08/2014

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, refer to 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	01/2009	Initial Release.
Rev. 2	04/2011	NOTE about USB OTG software/examples unavailability in the current MQX RTOS release added into the document.
Rev. 3	12/2011	Document re-written, 5 appendixes added.
Rev. 4	06/2012	Updated for MQX RTOS version 3.8.1 - see 4.3, "USB HDK Changes in MQX RTOS 3.8.1
Rev. 5	04/2013	Minor edits.
Rev. 6	10/2013	Updated content to reflect the switch from MQX types to C99 types.
Rev. 7	08/2014	Minor edits to Appendix C and Appendix D.

Chapter 1 Before You Begin

1.1 About This Book

This document describes the Freescale MQX™ USB Host Stack Architecture. This document does not distinguish between USB 1.1 and USB 2.0 information unless there is a difference between the two.

This document contains the following topics:

- Chapter 1, “Before You Begin”
- Chapter 2, “Get Familiar First”
- Chapter 3, “Design Overview”
- Chapter 4, “Developing Applications”
- Appendix A, “Working with the Software”
- Appendix B, “Human Interface Device (HID) Demo”
- Appendix D, “Virtual Communication (COM) Demo”
- Appendix C, “Mass Storage Device (MSD) Demo”
- Appendix E, “Audio Host Demo”

1.2 Acronyms and abbreviations

Table 1-1. Acronyms and abbreviations

Term	Description
API	Application Programming Interface
CDC	Communication Device Class
DCI	Device Controller Interface
HCI	Host Controller Interface
HID	Human Interface Device
MSD	Mass Storage Device
MSC	Mass Storage Class
PHD	Personal Healthcare Device
PHDC	Personal Healthcare Device Class
QOS	Quality Of Service
SCSI	Small Computer System Interface
USB	Universal Serial Bus

1.3 Reference Material

As recommendation consult the following reference material:

- *Universal Serial Bus Specification Revision 1.1*
- *Universal Serial Bus Specification Revision 2.0*
- *Freescale MQX RTOS USB Host API Reference Manual* (MQXUSBHOSTAPIRM)

Chapter 2 Get Familiar First

2.1 Introduction

Freescale MQX USB Host Stack operates at both low-speed and full speed. MQX USB Host Stack supports 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 Host stack in addition to providing useful information about developing and debugging USB applications. You can better understand the USB Host stack when you use this document together with the API documentation provided with the software suite. This document is targeted for firmware and software engineers who are familiar with the basic USB terms and who are directly involved in the product development on the USB Host Stack.

2.2 Software Suite

The Freescale MQX USB Host Stack software consists of the following:

- Class Layer API
- Device Framework
- Host Layer API

Class layer API, USB host class API, consists of the functions that can be used at the class level. This enables you to implement a new application based on pre-defined classes. This document describes four generic class implementations: Communication Device Class (CDC), Human Interface Device (HID), Mass Storage Class (MSD), Hub Class, and Audio Class API functions that are provided as part of the software suite. The API functions defined for these classes can be used to create applications.

Device Framework consists of functions that are used to support device requests which are common to all USB devices.

Host Layer API consists of the functions that can be used at the host level and support implementation on the class level.

For more details, see the Freescale MQX™ RTOS USB Host API Reference Manual (MQXUSBHOSTAPIRM).

2.3 Directory Structure

The following bullets display the complete tree of Freescale MQX USB Host Stack

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

The following table briefly explains the purpose of each of the directories in the source tree.

Table 2-1. Directory Structure

Directory Name	Descriptor
Usb	Root directory of the software stack
\lib\usbh	Output directory of the build
usb\host\build	*`*.mcp`* - for building the complete project
usb\host\examples	Host-side class-driver examples
usb\host\source\classes	Class drivers sources
usb\host\source	USB Host API sources
usb\host\source\host\khci	USB Host core drivers
Usb\host\rsource\rtos\mqx	RTOS layer source

Figure below shows the default directory structure.

NOTE

Figure below may not represent the latest tree but can provide an idea for the USB Host stack structure.

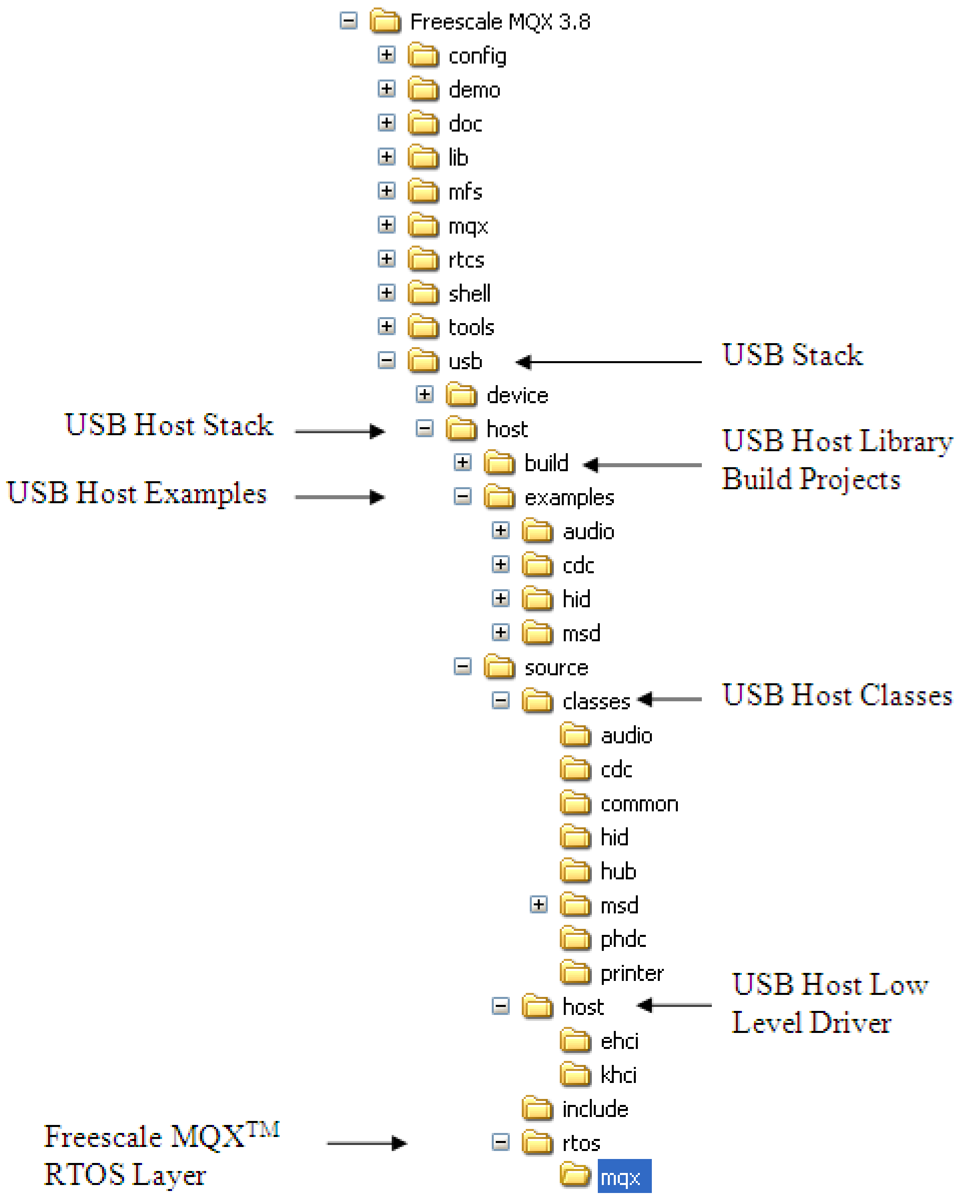


Figure 2-1. Default Directory Structure

Chapter 3 Design Overview

3.1 Design Goals

Freescale MQX USB Host stack has been designed with the following goals in mind.

3.1.1 Modularity

One of the goals of embedded software design is to achieve the target functionality with the minimum code size and with minimum components required for the product design. USB Host stack is designed to have flexible architecture so that undesired components can be taken away from the software suite. The USB Host stack team constatnly strives to add an extra functionality as a plug in module which can be completely eliminated at compile time.

3.1.2 Hardware Abstraction

One goal of the USB Host stack is to provide the true hardware independence in the application code. The USB Host stack is provided with the API routines that can detect the speed of the core and allow the application layer to make smart decisions based on the speed they are running under.

3.1.3 Performance

The USB Host stack has been designed to keep top performance, because API provides tight interaction with hardware. It is possible to eliminate the class drivers, operating system, or undesired routines from stack software to get the best performance possible on a target. The USB Host stack is designed to function under a single thread with minimum interrupt latency. When examining the code, notice that the USB Host stack is actually a two-layer stack with the constant goal of keeping the minimum layers and code lines required to achieve the abstraction goal.

3.2 A Target Design

A USB system is made up of a host and one or more devices that respond to the host request. The following figure shows the Target Design of the USB stack.

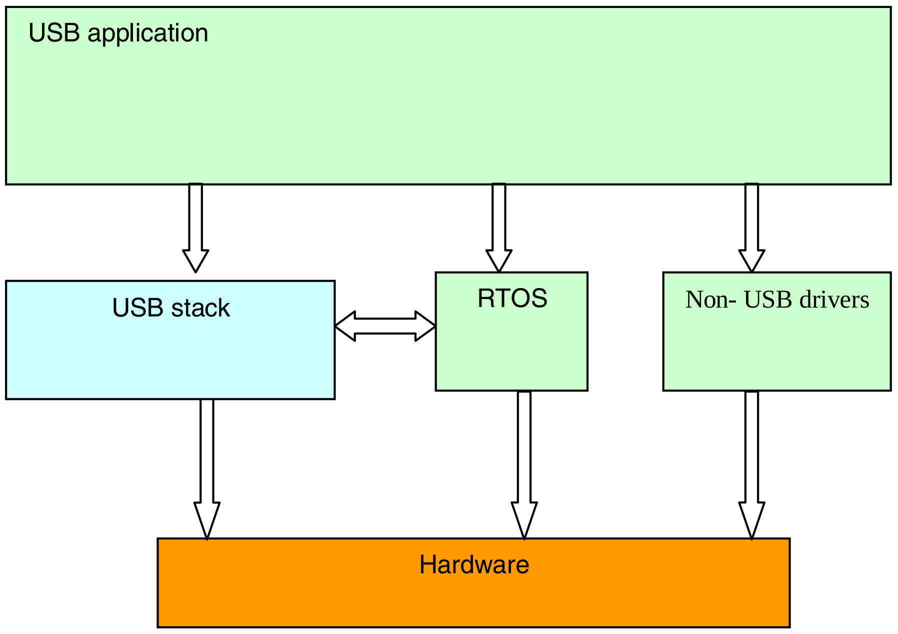


Figure 3-1. Target Design of USB Host Stack

The customer application can communicate with USB hardware using Freescale MQX USB stack. Figure 3-2 shows the stack design in more detail.

3.2.1 Complete USB Stack Diagram

The following figure shows the USB Stack Diagram.

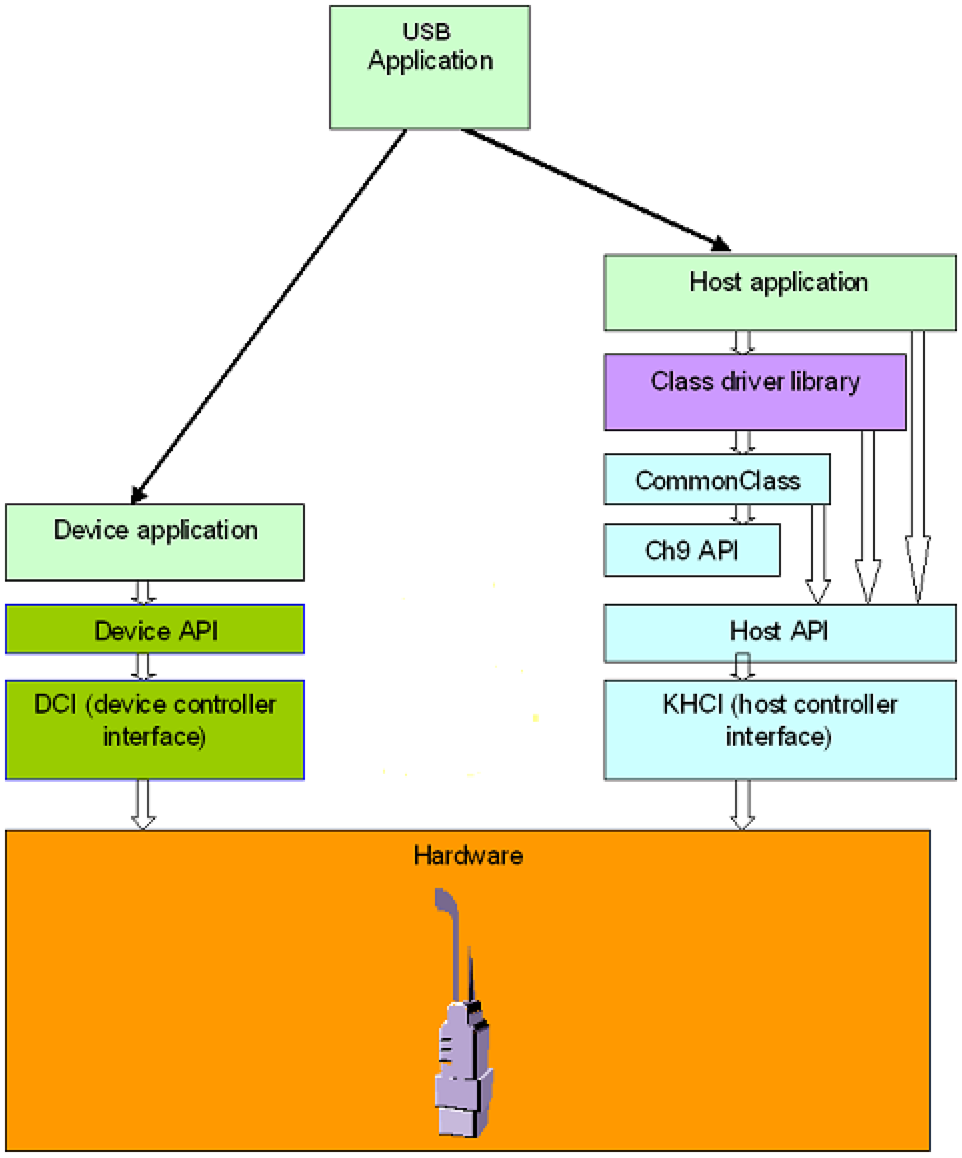


Figure 3-2. Complete USB Stack Diagram

3.3 Components Overview

3.3.1 Host Overview

The purpose of the Freescale MQX USB Host Stack is to provide an abstraction of the USB hardware controller core. A software application written using the host API can run on full-speed or low-speed core with no information about the hardware. In the USB, the host interacts with the device by using logical pipes. After the device is enumerated, a software application needs to be able to open and close the pipes.

After a pipe is opened with the device, the software application can queue transfers in either direction and is notified with the transfer result through an error status. In short, the communication between the host and the device is done by using logical pipes that are represented in the stack as pipe handles. Figure below shows the description of each of the blocks shown as part of the host API.

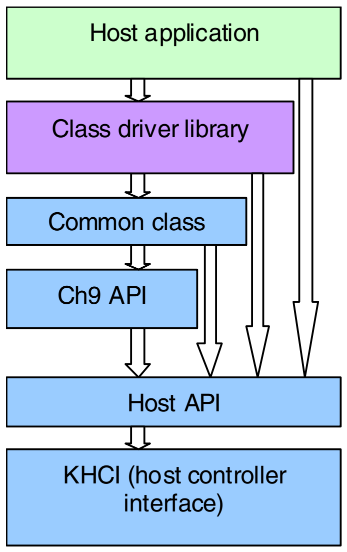


Figure 3-3. Freescale MQX USB Host Stack

3.3.1.1 Host Application

A host embedded application software, also called a device driver, is implemented for a target device or a class of device. Freescale MQX USB Host Stack includes examples for a few classes of the USB Host. Application software can use API routines to start and stop communication with the USB bus. It can suspend the bus or wait for a resume signal for the bus. See Chapter 4, "Developing Applications for more details.

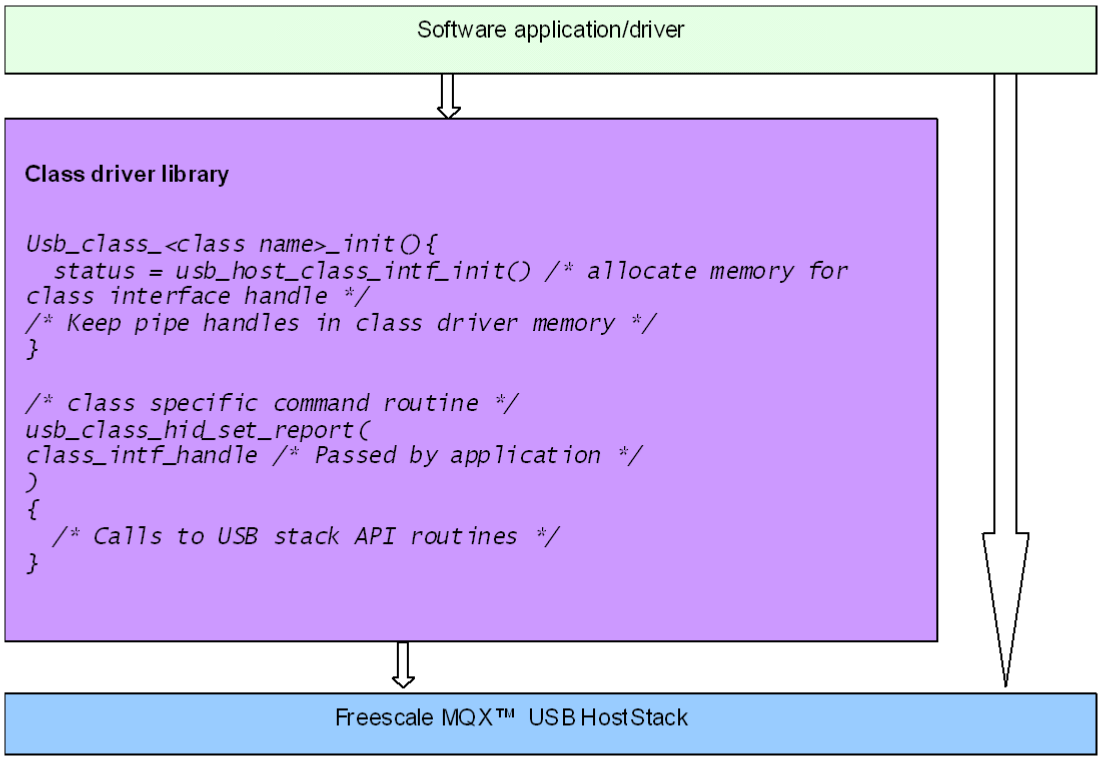
3.3.1.2 Class-Driver Library

The class-driver library is a set of wrapper routines that can be linked to the application. These routines implement standard functionality of the class of device as defined by the USB class specifications.
It is important to note that even though a class library can help in reducing some implementation effort at the application level, some USB devices do not implement class specifications, making it extremely difficult to write a single application code that can communicate with all devices. Storage devices mainly follow universal floppy interface (UFI) specifications making it easy to write an application that can communicate with most storage devices.

USB Host stack includes a class library for UFI commands that can be called to communicate with this kind of device. It is important to understand that a class-driver library is a set of wrapper routines written on a USB Host stack and does not necessarily have to be used.

The design of the class driver libraries in the USB Host stack follows a standard template. All libraries have an initialization routine called by the USB Host stack when a device of that particular class is plugged in. This routine allocates memory required for a class driver and constructs a structure called class handle.

Application software can receive the class handle when an interface is selected in the device. See Chapter 4, “Developing Applications. Once an interface is selected, applications can call class driver routines by using this handle and other necessary parameters. An implementation approach of class-specific routines is class-dependent and can vary completely from class to class. Figure 3-4 demonstrates the general design of all class-driver libraries.



```
Usb_class_<class name>_init(){
  status = usb_host_class_intf_init() /* allocate memory for
class interface handle */
/* Keep pipe handles in class driver memory */
}

/* class specific command routine */
usb_class_hid_set_report(
class_intf_handle /* Passed by application */
)
{
  /* Calls to USB stack API routines */
}
```

Figure 3-4. General Design of Class-Driver Libraries

3.3.1.3 Example of Mass-Storage Class-Driver Design

This section describes the driver design for the mass-storage class.

3.3.1.4 Architecture and Data Flow

All USB mass-storage commands are comprised of three phases:

- CBW (Command Block Wrapper)
- Data
- CSW (Command Status Wrapper)

For details about these phases, see a reference document about the USB mass-storage specifications. The mass-storage commands supported by the class-driver library are listed in the Mass Storage Class API documentation.

The Freescale MQX USB Host Stack Mass Storage Device class-driver library takes a command from an application and queues it in the local queue. It then starts with the CBW phase of the transfer followed by DATA and CSW phases. After the status phase is finished, it picks up the next transfer from the queue and repeats the same steps. It can make very high level calls on a UFI command set as read capacity or format drive and wait until a completion interrupt is received.

The CBW and CSW phase data are both described in two USB standard data structures:

- *CBW_STRUCT*
- *CSW_STRUCT*

All information concerning the mass-storage command, for example CBW and CSW structure pointer, data pointer, length, and so on are contained in the command structure *COMMAND_OBJECT_STRUCT*. Applications use this structure to queue the commands inside the class-driver library. If an application wants to send a vendor-specific call to a storage device, it must fill the fields of this structure and send it to the library by using a routine that can pass the command down to the USB.

All commands have a single-function entry point in the mass-storage API. However, all commands are mapped into a single function, using the *#defined* key word for code efficiency:

`usb_mass_ufi_generic() (see usb_mass_ufi.c and usb_mass_ufi.h).`

All data-buffer transmissions are executed using pointer and length parameters. There is no buffer copying across functions.

The interface between the mass-storage device class driver and the USB Host driver is **usb_host_send_data()** function. The parameters of this function are:

- The handle onto the USB Host peripheral.
- The handle onto the communication pipe with the mass-storage device.
- A structure describing the transfer parameter, size, and data pointer.

The objects in Figure 3-5 represent the following:

- The yellow arrows represent events originated in the USB Host driver.
- The black arrows represent movement to and from the mass-storage command queue.

- The boxes represent functions with actions taken and functions called.

Numbers one to ten represent the sequence of events or queue movements.

Functions with *_mass_* in their names are from the USB mass-storage library. Functions with *_host_* in their names are links to the USB host-driver library. Look up the functions in the class-driver library source to better understand the code flow.

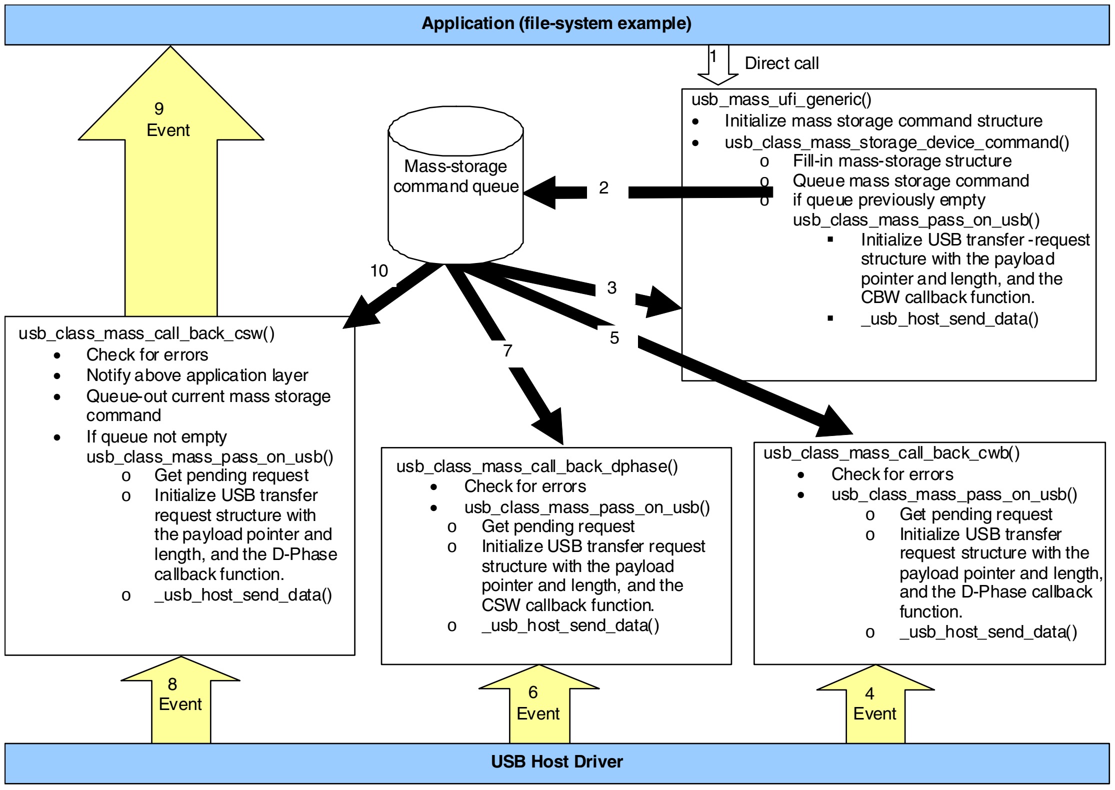

Application (file-system example)
1
Direct call
usb_mass_ufi_generic()
Initialize mass storage command structure
usb_class_mass_storage_device_command()
Fill-in mass-storage structure
Queue mass storage command
if queue previously empty
usb_class_mass_pass_on_usb()
Initialize USB transfer -request structure with the payload pointer and length, and the CBW callback function.
_usb_host_send_data()
9
Event
Mass-storage command queue
2
10
3
5
7
usb_class_mass_call_back_csw()
Check for errors
Notify above application layer
Queue-out current mass storage command
If queue not empty
usb_class_mass_pass_on_usb()
Get pending request
Initialize USB transfer request structure with the payload pointer and length, and the D-Phase callback function.
_usb_host_send_data()
usb_class_mass_call_back_dphase()
Check for errors
usb_class_mass_pass_on_usb()
Get pending request
Initialize USB transfer request structure with the payload pointer and length, and the CSW callback function.
_usb_host_send_data()
usb_class_mass_call_back_cwb()
Check for errors
usb_class_mass_pass_on_usb()
Get pending request
Initialize USB transfer request structure with the payload pointer and length, and the D-Phase callback function.
_usb_host_send_data()
8
Event
6
Event
4
Event
USB Host Driver

Figure 3-5. Mass-Storage-Driver Code Flow

3.3.1.5 Common-Class API

Common-class API is a layer of routines that implements the common-class specification of the USB and an operating system level abstraction of the USB system. This layer interacts with the host API layer functions. By looking at the API document, it is difficult to say which routines belong to this layer. It is a deliberate design attempt to reuse routines and minimize the code size.

Routines inside the common-class layer use the fact that, in USB, all devices are enumerated with the same sequence of commands. When an USB device is plugged into the host hardware, it generates an interrupt, and lower-level drivers call the common-class layer to start the device. Routines inside the common-class layer allocate memory, assign the USB address, and enumerate the device. After the device descriptors are identified, the common-class layer searches for applications that are registered for the class or device plugged in. If a match is found, a callback is generated for the application to start communicating with the device. See Chapter 4, “Developing Applications, for information about how an application handles a plugged-in device. Figure below illustrates how the device plugin works.

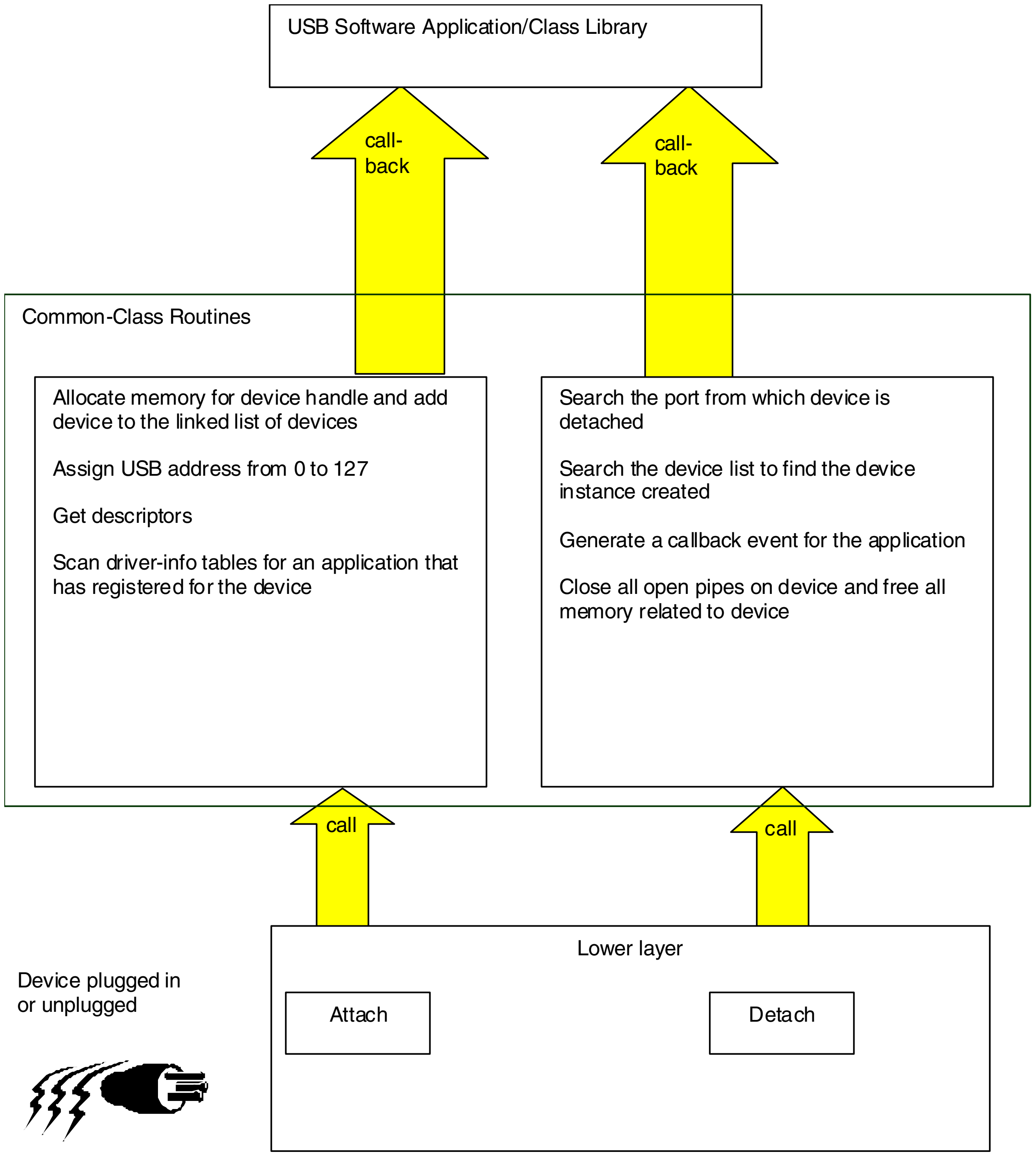

USB Software Application/Class Library
call-back
call-back
Common-Class Routines
Allocate memory for device handle and add device to the linked list of devices
Assign USB address from 0 to 127
Get descriptors
Scan driver-info tables for an application that has registered for the device
Search the port from which device is detached
Search the device list to find the device instance created
Generate a callback event for the application
Close all open pipes on device and free all memory related to device
call
call
Lower layer
Attach
Detach
Device plugged in or unplugged

Figure 3-6. How Devices are Attached and Detached

3.3.1.6 USB Chapter 9 API

Chapter 9 in the USB specification document is dedicated to standard command protocol implemented by all USB devices. All USB devices are required to respond to a certain set of requests from the host. This API is a low-level API that implements all USB commands mentioned in Chapter 9. All customer applications can be written to use only this API without the common-class API or class libraries.

Chapter 9 commands are outside the scope of this document and require a detailed familiarity with the USB protocol and higher-level abstraction of USB devices. In conclusion, the following are some of the example routines that are implemented by this API. See source code for implementation details.

- **usb_host_ch9_dev_req ()** — For sending control pipe setup packets.
- **_usb_host_ch9_clear_feature()** — For a clear feature USB command.
- **_usb_host_ch9_get_descriptor ()** — For receiving descriptors from device.

3.3.1.7 Host API

The host API is a hardware abstraction layer of the Freescale MQX USB Host stack. This layer implements routines independent of underlying USB controllers. For example, **usb_host_init()** initializes the host controller by calling the proper hardware-dependent routine. Similarly, **usb_host_shutdown()** shuts down the proper hardware host controller. The following are the architectural functions implemented by this layer.

- This layer allocates pipes from a pool of pre-allocated pipe memory structures when **usb_host_open_pipe ()** is called.
- This layer maintains a list of all transfers pending per pipe used in freeing all memory when a pipe is closed.
- This layer maintains a link list of all services (callbacks) registered with the stack. When a specific hardware event, such as attach or detach, occurs, it generates a callback and informs the upper layers.
- This layer provides routines to cancel USB transfers by calling hardware-dependent functions.
- This layer provides other hardware-control routines such as the ability to shutdown the controller, suspend the bus, and so on.

To better understand the source inside the API layer, read the API routine and trace it to the hardware drivers.

3.3.1.8 HCI (Host Controller Interface)

HCI is a completely hardware-dependent set of routines responsible for queuing and processing USB transfers and searching for hardware events. To understand the source of this layer, you have to understand the hardware.

The MQX RTOS implements drivers for two kinds of HCIs:

- KHCI - used in ColdFire V2 and some Kinetis devices.
- EHCI - used in high-end ColdFire V4 and some Kinetis and PowerPC devices.

Chapter 4 Developing Applications

4.1 Compiling Freescale MQX USB Host Stack

4.1.1 Why Rebuild USB Host Stack

It is necessary to rebuild the USB Host stack if any of the following is done:

- Change compiler options (optimization level)
- Change USB Host stack compile-time configuration options
- Incorporate changes made to the USB Host source code

CAUTION

It is not recommend to modify USB Host stack data structures. If the USB Host stack data structures are modified, some of the components in the Precise Solution™ Host Tools family of host software-development tools may not perform correctly. Modify USB Host stack data structures only if you have experience with the USB Host stack.

4.1.1.1 Before Beginning

Before rebuilding the USB Host stack do the following:

- Read the MQX RTOS User's Guide to get MQX RTOS rebuild instructions. A similar concept also applies to the USB Host stack.
- Read the MQX RTOS Release Notes that accompany Freescale MQX RTOS to obtain information specific to target environment and hardware.
- Have the required tools for target environment:
  - — compiler
  - — assembler
  - — linker
- Get familiarized with the USB Host stack directory structure and re-build instructions, as described in the Release Notes and the instructions provided in the following sections.

4.1.1.2 USB Directory Structure

The following figure shows the MQX USB Host Stack directory structure.

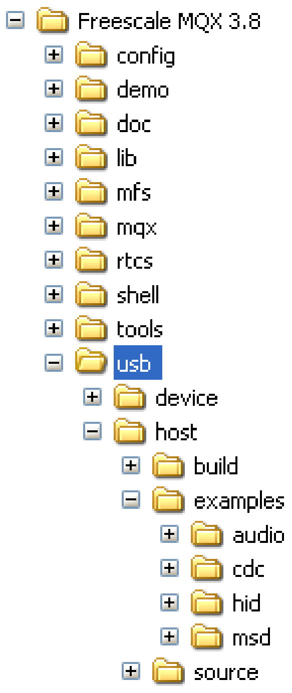


Figure 4-1. MQX USB Host Stack directory structure

The table below describes the USB Host stack directory structure.

config	The main configuration directory
<board>	Board-specific directory which contains the main configuration file (user_config.h)
usb\host	Root directory for USB Host stack within the Freescale MQX distribution
\build	
\<compiler>	Development tool-specific build files (project files)
\examples	
\example	Source files (.c) for the example and the example's build project.
\source	All USB Host stack source code files
\lib	
\<board>.<comp>\usb	USB Host stack library files built for hardware and environment

4.1.1.3 USB Host Stack Build Projects in Freescale MQX RTOS

The USB Host stack build project is constructed just as the other core library projects included in Freescale MQX RTOS. The build project for a given development environment, for example CodeWarrior, is located in the usb\host\build\<compiler> directory. The USB Host stack code is not specific to any particular board or to a processor derivative. Instead, a separate USB Host stack build project exists for each supported board. The resulting library file is built into a board-specific output directory in lib\<board>.<compiler>.

The reason that the board-independent code is built into the board-specific output directory is that it can be configured for each board separately. The compile time user-configuration file is taken from a board-specific directory config\<board>. The user may want to build the resulting library code differently for two different boards. See the MQX RTOS User's Guide for more details.

4.1.1.3.1 Post-Build Processing

All USB Host stack build projects are configured to generate the resulting binary library file in the top-level lib\<board>.<compiler>\usb directory. For example, the CodeWarrior libraries for the M52259EVB board are built into the lib\m52259evb.cw\usb directory.

The USB Host stack build project is also set up to execute a post-build batch file that copies all the public header files to the destination directory. This makes the output \lib directory the only place accessed by the application code. The MQX applications projects that need to use the USB Host stack services do not need to make any reference to the USB Host stack source tree.

4.1.1.3.2 Build Targets

Development tools enable multiple build configurations, so called build targets. All projects in the Freescale MQX USB Host stack contain at least two build targets:

- Debug Target — Compiler optimizations are set low to enable easy debugging. Libraries built using this target are named "_d" postfix. For example, lib\m52259evb.cw\usb\ usb_hdk_d.a.
- Release Target — Compiler optimizations are set to maximum to achieve the smallest code size and fast execution. The resulting code is hard to debug. The generated library name does not get any postfix. For example, lib\m52259evb.cw\usb\ usb_hdk.a.

4.1.1.4 Rebuilding Freescale MQX USB Host Stack

Rebuilding the MQX USB Host stack library is a simple task that involves opening the proper build project in the development environment and building it. Do not forget to select either the proper build target to build, or to build all targets.

For specific information about rebuilding MQX USB Host stack and the example applications, see Freescale MQX RTOS Release Notes.

4.2 Host Applications

The following steps are described to achieve the host functionality.

4.2.1 Background

In the USB system, the host software controls the bus and talks to the target devices under the rules defined by the specification. A device is represented by a configuration which is a collection of one or more interfaces. Each interface is comprised of one or more endpoints. Each endpoint is represented as a logical pipe from the application software perspective.

The host application software registers for services with the USB Host stack and describes the callback routines inside the driver info table. When a USB device is connected, the USB Host stack driver enumerates the device automatically and generates interrupts for the application software. One interrupt per interface is generated on the connected device. If the application likes to talk to an interface, it can select that interface and receive the pipe handles for all the end points. See the host API document with the source code example to see what routines are called to find pipe handles. After the application software receives the pipe handles, it can start the communication with the pipes. If the software likes to talk to another interface or configuration, it can call the appropriate routines again to select another interface.

The USB Host stack consists of a few lines of code before one starts communication with the USB device. Examples on the USB stack can be written with a host API only. However, most examples supplied with the stack are written using class drivers. Class drivers work with the host API as a supplement to the functionality. It makes it easy to achieve a target functionality, see example sources for details, without the hassle of dealing with implementation of standard routines. The following code steps are taken inside a host application driver for any specific device.

4.2.2 Create a Project

Perform these steps to develop a new application:

1. Create a new application directory under /host/examples directory. The new application directory is used to make the new application.

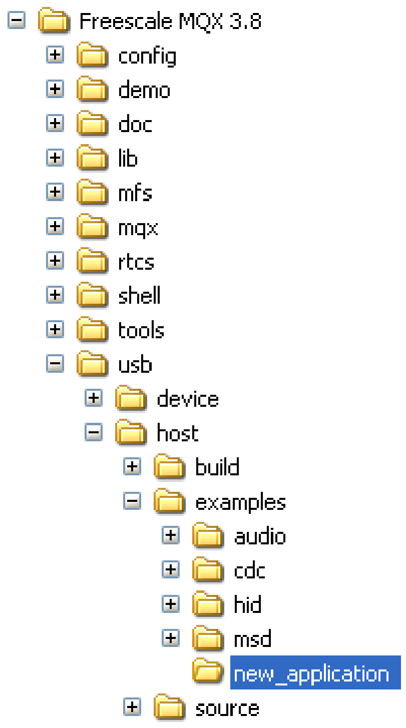


Figure 4-2. Create directory for application

2. Copy the usb_classes.h file from similar existing applications.

   usb_classes.h is used to define classes that are used in the application.

   Example: If you want to create an application that uses HID and HUB, you can define it as follows:

```
#define USBCLASS_INC_HID
#define USBCLASS_INC_HUB
```

3. Create a directory where the project files for the new application can be created.

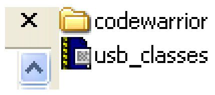


Figure 4-3. Create CodeWarrior project and usb_classes.h file

4. Create new files for making the main application function and callback.

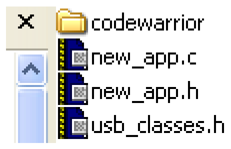


Figure 4-4. Create header and code file

- The new_app.h file contains application types and definitions:

  Examples:

  Define states of device :

```
/* Used to initialize USB controller */
#define MAX_FRAME_SIZE           1024
#define HOST_CONTROLLER_NUMBER      0
#define HID_BUFFER_SIZE                    4
#define  USB_DEVICE_IDLE                   (0)
#define  USB_DEVICE_ATTACHED               (1)
#define  USB_DEVICE_CONFIGURED             (2)
#define  USB_DEVICE_SET_INTERFACE_STARTED  (3)
#define  USB_DEVICE_INTERFACED             (4)
#define  USB_DEVICE_SETTING_PROTOCOL       (5)
#define  USB_DEVICE_INUSE                  (6)
#define  USB_DEVICE_DETACHED               (7)
#define  USB_DEVICE_OTHER                  (8)

/*
** Following structs contain all states and pointers
** used by the application to control/operate devices.
*/

typedef struct device_struct {
   uint32_t                         DEV_STATE;  /* Attach/detach state */
   _usb_device_instance_handle      DEV_HANDLE;
   _usb_interface_descriptor_handle INTF_HANDLE;
   CLASS_CALL_STRUCT                CLASS_INTF; /* Class-specific info */
} DEVICE_STRUCT,  * DEVICE_STRUCT_PTR;

/* Alphabetical list of Function Prototypes */

#ifdef __cplusplus
extern "C" {
#endif

void usb_host_hid_recv_callback(_usb_pipe_handle, void *, unsigned char *,
uint32_t,
   uint32_t);
void usb_host_hid_ctrl_callback(_usb_pipe_handle, void *, unsigned char *,
uint32_t,
   uint32_t);
void usb_host_hid_mouse_event(_usb_device_instance_handle,
   _usb_interface_descriptor_handle, uint32_t);

#ifdef __cplusplus
}
#endif
```

- The new_app.c file contains driver informations, callback functions and event functions, and the main function.

4.2.3 Define a Driver Info Table

A driver info table defines devices that are supported and handled by this target application. This table defines the PID, VID, class, and subclass of the USB device. The Host stack generates an attached callback when a device matches this table entry. The application can now communicate to the device. The following structure defines one member of the table. If the Vendor-Product pair does not match a device, Class -Subclass, and Protocol is checked to match. Use 0xFF in SubClass and Protocol struct members to match any SubClass/Protocol.

```
/* Information for one class or device driver */
typedef struct driver_info
{
   uint8_t          idVendor[2];      /* Vendor ID per USB-IF */
   uint8_t          idProduct[2];     /* Product ID per manufacturer */
   uint8_t          bDeviceClass;     /* Class code, 0xFF if any */
   uint8_t          bDeviceSubClass;  /* Sub-Class code, 0xFF if any */
   uint8_t          bDeviceProtocol;  /* Protocol, 0xFF if any */
   uint8_t          reserved;         /* Alignment padding */
event_callback attach_call;          /* event callback function*/
} USB_HOST_DRIVER_INFO, * USB_HOST_DRIVER_INFO_PTR;
```

The following is a sample driver info table. See the example source code for samples. Notice that the following table defines all HID MOUSE devices that are boot subclass. A terminating NULL entry in the table is always created for search end.

Since, in the HID MOUSE application, two classes are used (HID and HUB), the DriverInfoTable variables has 3 elements. There are 2 event callback functions for 2 classes: **usb_host_hid_keyboard_event** for HID class and **usb_host_hub_device_event** for HUB class.

```
/* Table of driver capabilities this application wants to use */
static USB_HOST_DRIVER_INFO DriverInfoTable[] = {
{
        {0x00, 0x00},       /* Vendor ID per USB-IF             */
        {0x00, 0x00},       /* Product ID per manufacturer      */
        USB_CLASS_HID,      /* Class code                       */
        USB_SUBCLASS_HID_BOOT, /* Sub-Class code                   */
        USB_PROTOCOL_HID_KEYBOARD, /* Protocol                         */
        0,                  /* Reserved                         */
        <add the name of your event callback function here>
        usb_host_hid_keyboard_event /* Application call back function   */
},
        /* USB 1.1 hub */
{
        {0x00, 0x00},       /* Vendor ID per USB-IF             */
        {0x00, 0x00},       /* Product ID per manufacturer      */
        USB_CLASS_HUB,      /* Class code                       */
        USB_SUBCLASS_HUB_NONE, /* Sub-Class code                   */
        USB_PROTOCOL_HUB_LS, /* Protocol                         */
        0,                  /* Reserved                         */
        <add the name of your event callback function here>
```

```
            usb_host_hub_device_event /* Application call back function   */
    },
    {
            {0x00, 0x00},       /* All-zero entry terminates        */
            {0x00, 0x00},       /* driver info list.                */
            0,
            0,
            0,
            0,
            NULL},
    }
```

4.2.4 Main application function flow

In the main application function, it is necessary to follow these steps:

1. Initializing hardware
2. Initializing the Host Controller
3. Registering service

Call tasks in a forever loop.

4.2.4.1 Initializing the Host Controller

The first step, which is required to act as a host, is to initialize the stack in a host mode. This allows the stack to install a host interrupt handler and initialize the necessary memory required to run the stack. The following example illustrates this:

```
error = _usb_host_init(0, 1024, &host_handle);
if (error != USB_OK)
{
      printf("\nUSB Host Initialization failed. Error: %x", error);
      fflush(stdout);
}
```

Second argument (1024 in the above example), in the code above, is the size of a periodic frame list. Full speed customers can ignore the argument.

4.2.4.2 Register Services

Once the host is initialized, the USB Host stack is ready to provide services. An application can register for services as documented in the host API document. The host API document allows the application to register for an attached service, but applications that are using the driver info table do not need to register for this service because the driver info table already registers a callback routine. The following example shows how to register for a service on the host stack:

```
error = _usb_host_register_service (host_handle,
USB_SERVICE_HOST_RESUME,
App_process_host_resume);
```

This code registers a routine called app_process_host_resume() when the USB host controller resumes operating after a suspend. See the USB specifications about how to suspend and resume work under the USB Host.

NOTE

Some examples do not register for services because the driver info table has already registered the essential routine for the attached service.

4.2.4.3 Enumeration Process of a Device

After the software has registered the driver info table and other services, it is ready to handle devices. In the USB Host stack, users do not have to write any enumeration code. As soon as the device is connected to the host controller, the USB Host stack enumerates the device and finds the number of interfaces supported. Additionally, it scans the registered driver info tables and finds which application has registered for the device for each interface. It provides a callback if the device criteria matches the table. The application software has to choose the interface. The following is a sample code that does this:

```
void usb_host_hid_mouse_event
   (
      /* [IN] pointer to device instance */
      _usb_device_instance_handle      dev_handle,

      /* [IN] pointer to interface descriptor */
      _usb_interface_descriptor_handle intf_handle,

      /* [IN] code number for event causing callback */
      uint32_t                          event_code
   )
{ /* Body */
   INTERFACE_DESCRIPTOR_PTR   intf_ptr =
      (INTERFACE_DESCRIPTOR_PTR)intf_handle;

   fflush(stdout);
   switch (event_code) {
      case USB_CONFIG_EVENT:
         /* Drop through into attach, same processing */
      case USB_ATTACH_EVENT:
         fflush(stdout);
         printf("State = %d", hid_device.DEV_STATE);
         printf("  Class = %d", intf_ptr->bInterfaceClass);
         printf("  SubClass = %d", intf_ptr->bInterfaceSubClass);
         printf("  Protocol = %d\n", intf_ptr->bInterfaceProtocol);
         fflush(stdout);

         if (hid_device.DEV_STATE == USB_DEVICE_IDLE) {
            hid_device.DEV_HANDLE = dev_handle;
            hid_device.INTF_HANDLE = intf_handle;
            hid_device.DEV_STATE = USB_DEVICE_ATTACHED;
         } else {
            printf("HID device already attached\n");
            fflush(stdout);
         } /* Endif */
         break;
```

Notice that in the code above, the application matched the first call to the USB_ATTACH_EVENT() and stored the interface handle under a local variable called `hid_device.INTF_HANDLE`. It also changed the state of the program to `USB_DEVICE_ATTACHED`.

4.2.4.4 Selecting an Interface on a Device

If the interface handle has been obtained, application software can select the interface and retrieve pipe handles. The following code demonstrates this procedure:

```
case USB_DEVICE_ATTACHED:
   printf("Mouse device attached\n");
   hid_device.DEV_STATE = USB_DEVICE_SET_INTERFACE_STARTED;
   status = _usb_hostdev_select_interface(hid_device.DEV_HANDLE,
      hid_device.INTF_HANDLE, (void *)&hid_device.CLASS_INTF);
     if (status != USB_OK) {
      printf("\nError in _usb_hostdev_select_interface: %x", status);
      fflush(stdout);
      exit(1);
   } /* Endif */
   break;
```

As internal information, usb_hostdev_select_interface caused the stack to allocate memory and do the necessary preparation to start communicating with this device. This routine opens logical pipes and allocates bandwidths on periodic pipes. This allocation of bandwidths can be time consuming under complex algorithms.

4.2.4.5 Retrieving and Storing Pipe Handles

If the interface has been selected, pipe handles can be retrieved by calling, as shown in this example:

```
pipe = _usb_hostdev_find_pipe_handle(hid_device.DEV_HANDLE,
               hid_device.INTF_HANDLE, USB_INTERRUPT_PIPE, USB_RECV);
```

In this code, a pipe is a memory pointer that stores the handle (see code example for details). Notice that this routine specified the type of the pipe retrieved. The code shows how to communicate to a mouse that has an interrupt pipe to obtain the pipe handle for the interrupt pipe.

4.2.4.6 Sending/ Receiving Data to/ from Device

The USB packet transfers on USB software functions by using transfer requests (TR). It is a similar concept to the URB for Windows and Linux. For Windows, drivers keep sending URBs down the stack and waiting for events or callbacks for USB completion. There is one callback or event per URB completion. The USB stack concept is the same except that fields inside a TR can be different. A TR is a memory structure that describes a transfer in its entirety. The USB stack provides a helper routine called usb_hostdev_tr_init() that can be used to initialize a TR. Every TR down the stack has a unique number assigned by the tr_init() routine. The following code example shows how this routine is called:

```
usb_hostdev_tr_init(&tr, usb_host_hid_recv_callback, <parameter>);
```

This routine takes the `tr` pointer to the structure that needs to be initialized and the name of the callback routine that is called when this TR is complete. An additional parameter can be supplied that is called back when TR completes. Unlike PC based systems, the embedded systems memory is limited and, therefore, a recommended practice is to reuse the TR that is supplied. Applications can keep a few TRs pending and reuse old ones after completed. See the code example for details.

After TR is initialized and pipe handle available, it is easy to send and receive data to the device. USB devices that use periodic data need a periodic call to send to receive data. It is recommended to use operating system timers to ensure that a receive or send data call is done in a timely manner so that packets to and from the device are not lost. These details are USB driver design details and are outside the scope of this document. The following code provides an example how a receive data is done.

```
status = _usb_host_recv_data(host_handle, pipe, &tr);
```

4.2.4.7 Other Host Functions

The USB Stack comes with a wide set of routines that can be used to exploit the functionality available. There are routines available to open pipes, close pipes, get frame numbers, micro frame numbers, or even shut down the stack. These routines are obvious by their names and many are used at various places in the code. For example, _usb_host_bus_control() routine can be used to suspend the USB bus at any time under the software control. Similarly, usbhost_shutdown() can be called to shut down the host stack and free all memory. This routine ensures that all pipes are closed and that the memory is freed by the stack. These functions can be used on as needed basis. As a suggestion, search the examples that use some of these routines and copy the code if needed.

4.3 USB HDK Changes in MQX RTOS 3.8.1

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

The common part of USB stacks was moved into *usb/common* folder. In the 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 the following API changes if porting from previous MQX versions:

BSP:

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

USB-HDK:

- `USB_STATUS _usb_host_driver_install(usb_host_if_struct *)` API changed
- `USB_STATUS _usb_host_init(usb_host_if_struct *)` API changed

- `void  (_CODE_PTR_tr_callback)(void *, void *, unsigned char *, uint32_t, USB_STATUS)` API changed
- the `TR_INIT_PARAM_STRUCT` has changed and contains generic superstructure at the top

In the file *source/bsp/<bsp_name>/<board>.h* you can find definition of the USB host controller for a given board, for example:

```
#define USBCFG_DEFAULT_HOST_CONTROLLER    (&_bsp_usb_host_ehci0_if)
```

You can use these macros in the `_usb_host_driver_install` and `_usb_host_init` functions.

4.3.1 Migration steps for USB host application

1. Application project - add compiler search path (*lib/<board>.<comp>/usb*) to make common header files (*usb.h, ...*) available for 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 <hostapi.h> /* if you want to use USB-HDK */
/* Additionally, include USB host class driver header files… */
```

3. Change the way the low level driver is installed:

```
res = _usb_host_driver_install(USBCFG_DEFAULT_HOST_CONTROLLER);
```

   ... instead of legacy `_usb_host_driver_install` which used additional arguments.

4. Change the low level driver initialization to:

```
res = _usb_host_init(USBCFG_DEFAULT_HOST_CONTROLLER, &host_handle);
```

5. Since the `tr_callback` API has changed, you should change the last parameter of the callback functions from `uint32_t` to `USB_STATUS`. This is not a real issue now as the `USB_STATUS` is defined as `uint32_t`, however, it may cause compatibility issues in the future.

6. If a USB host application explicitly prepares transfers to be sent, it should use the variable of the type `TR_INIT_PARAM_STRUCT` and change the way its members are accessed after calling `usb_hostdev_tr_init` function. As mentioned before, this structure has changed. Add "G" superstructure to the members of `TR_INIT_PARAM_STRUCT`. Use the following:

```
tr.G.RX_BUFFER, tr.G.RX_LENGTH, tr.G.TX_BUFFER, tr.G.TX_LENGTH
```

7. A known issue in MK70F KHCI controller occurs when you use SRAM memory for your buffers. It is strongly recommended that you use `_usb_hostdev_get_buffer(_usb_device_instance_handle, uint32_t, void **)` function to get a buffer allocated in the correct way and automatically deallocated when the device is unplugged. Allocate the buffer for a device plugged into the host:

```
    res  = _usb_hostdev_get_buffer(handle, size, &buffer);
```

Appendix A Working with the Software

A.1 Introduction

This chapter gives you insight about using applications in MQX USB Host Stack software. The following sections are described in this chapter:

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

Knowledge of CodeWarrior IDE will be helpful to understand this section. While reading this chapter, practice the steps mentioned.

A.1.1 Preparing the Setup

A.1.1.1 Hardware Setup

Make the connections as shown in Figure A-1.

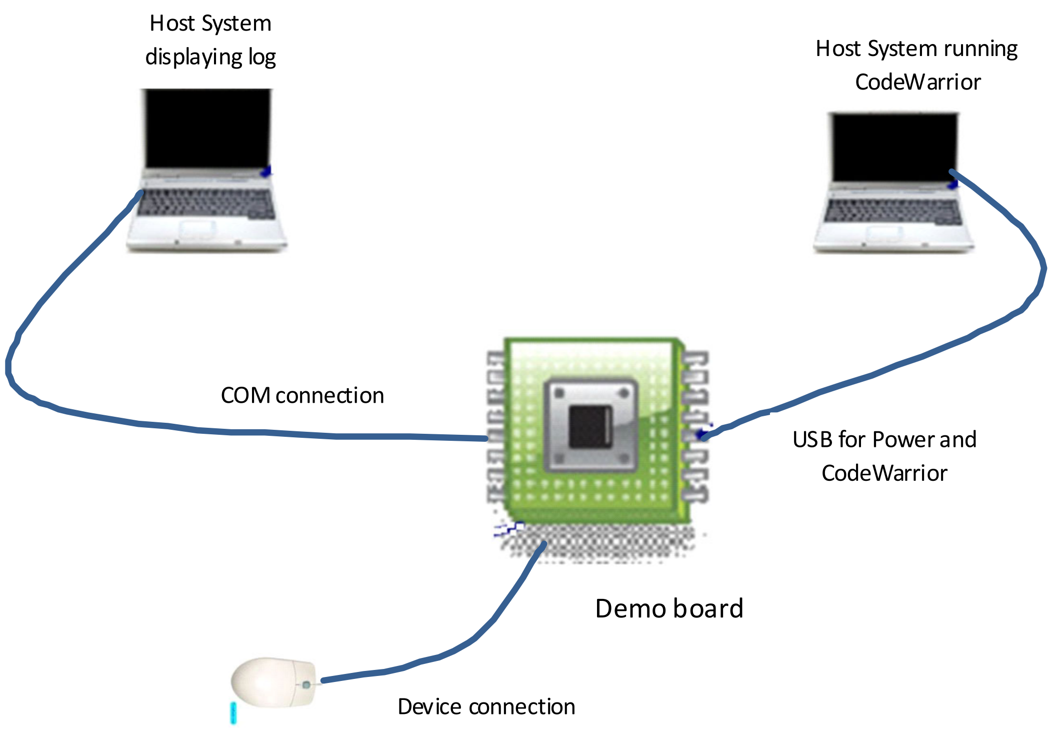

Host System displaying log
Host System running CodeWarrior
COM connection
USB for Power and CodeWarrior
Demo board
Device connection

Figure A-1. ColdFire V1 USB Setup

— Make the first USB connection between the personal computer, where the software is installed, and the demo board, where you'd like to run the application. This connection is required to provide power to the board and download the image to the flash.

— Make the second connection between the demo board and the personal computer to display the log of demo.

— Make the third connection between the device and the demo board.

A.1.2 Building the Application

The software was built using CodeWarrior 6.3 and, therefore, contains application project files that can be used to build the project.

Before starting the process of building the project, make sure CodeWarrior 6.3 is installed on your computer.

To build ColdFire V1 project:

1. Navigate to the project file and open xxx_m51jmevb.mcp project file in CodeWarrior IDE.

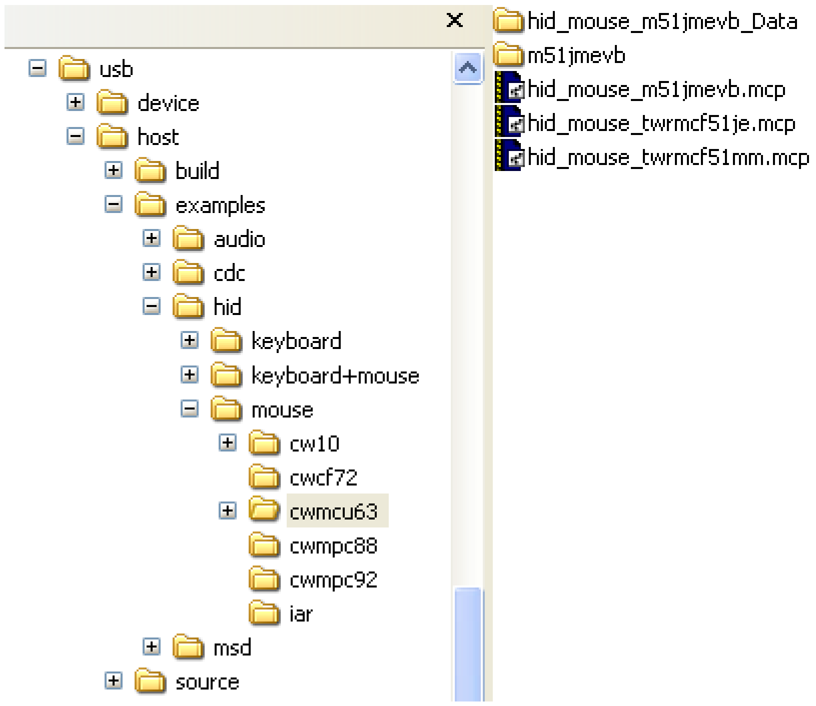
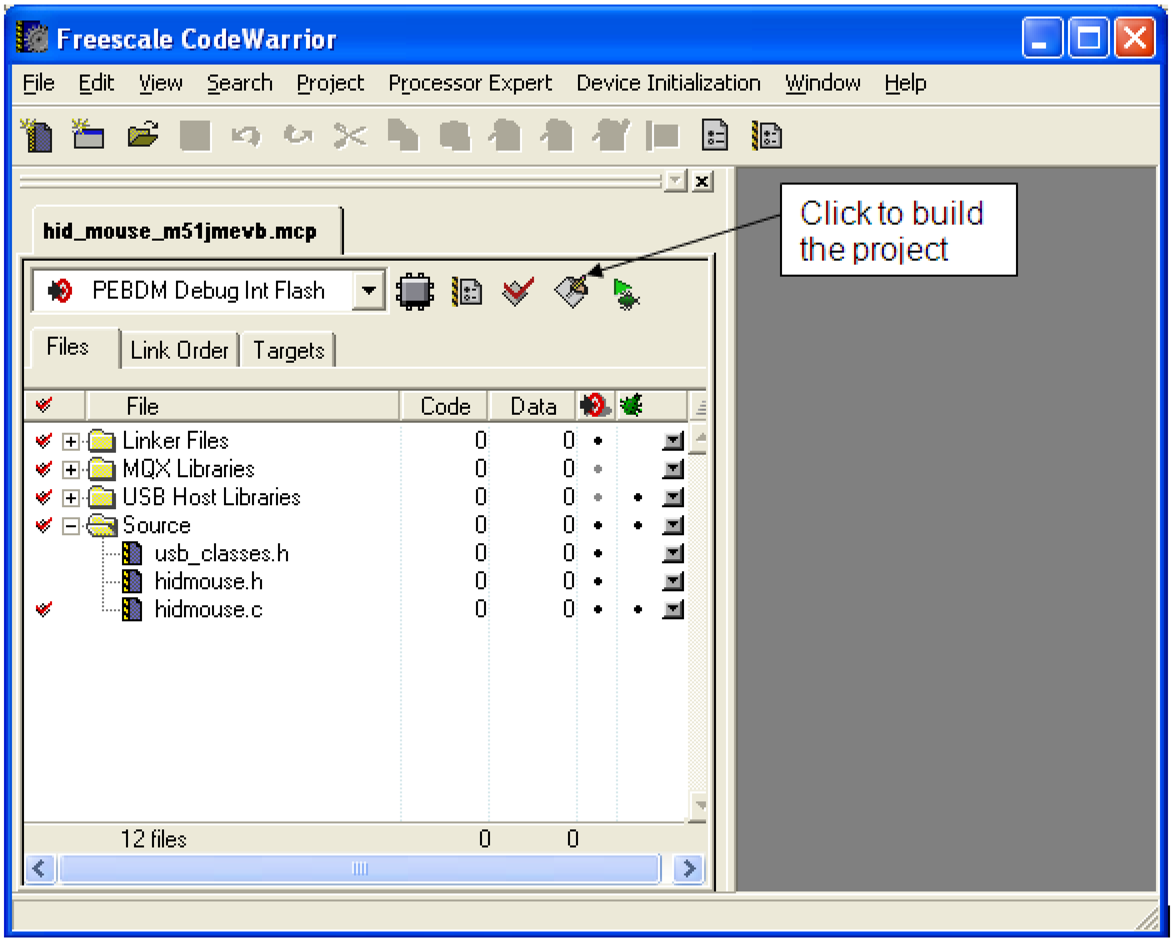


Figure A-2. Open xxx_m51jmevb.mcp Project File

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


Freescale CodeWarrior
File Edit View Search Project Processor Expert Device Initialization Window Help
hid_mouse_m51jmevb.mcp
PEBDM Debug Int Flash
Files Link Order Targets
Click to build the project

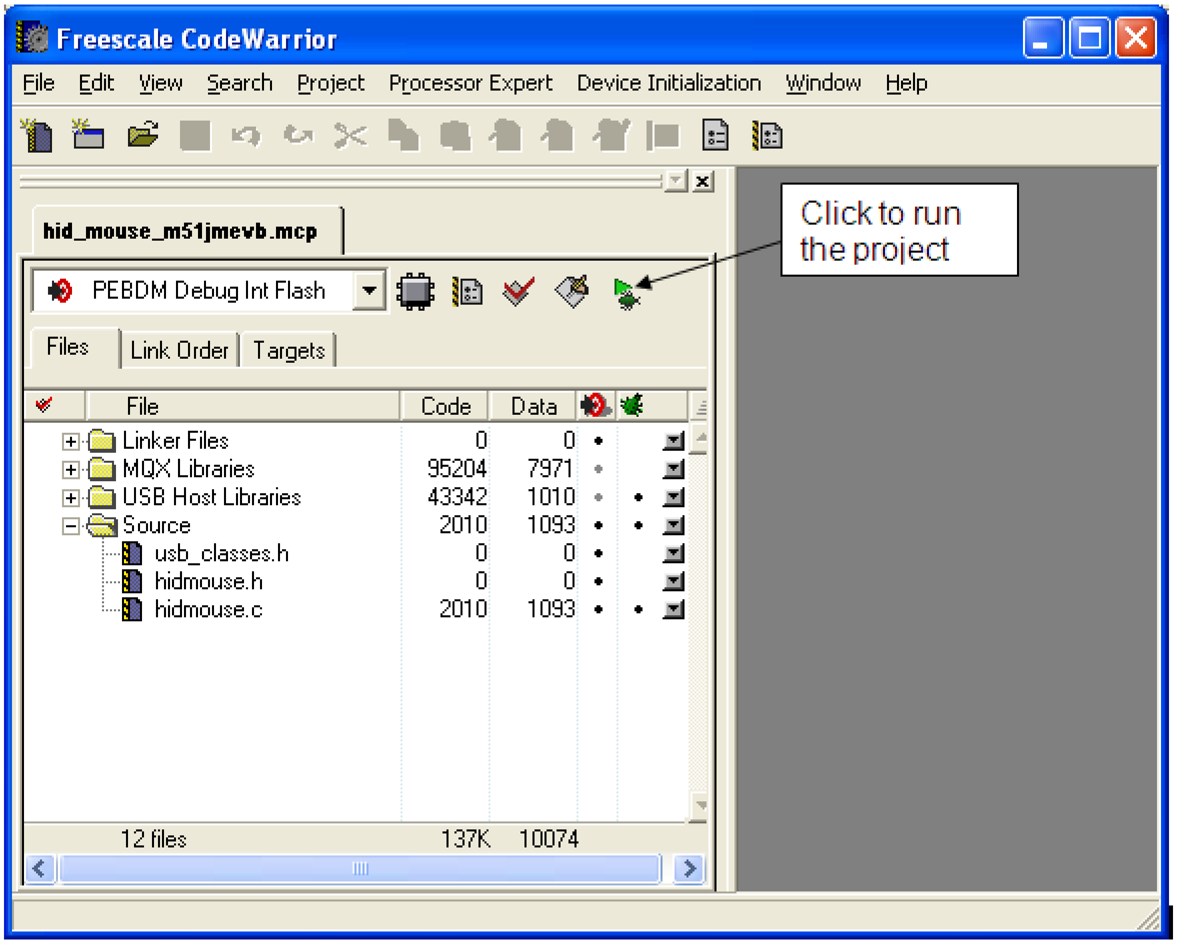
✓	File	Code	Data		
✓	Linker Files	0	0	•	
✓	MQX Libraries	0	0	•	
✓	USB Host Libraries	0	0	•	•
✓	Source	0	0	•	•
	usb_classes.h	0	0	•	
	hidmouse.h	0	0	•	
✓	hidmouse.c	0	0	•	•
	12 files	0	0		

Figure A-3. Build Project

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

A.1.3 Running the application

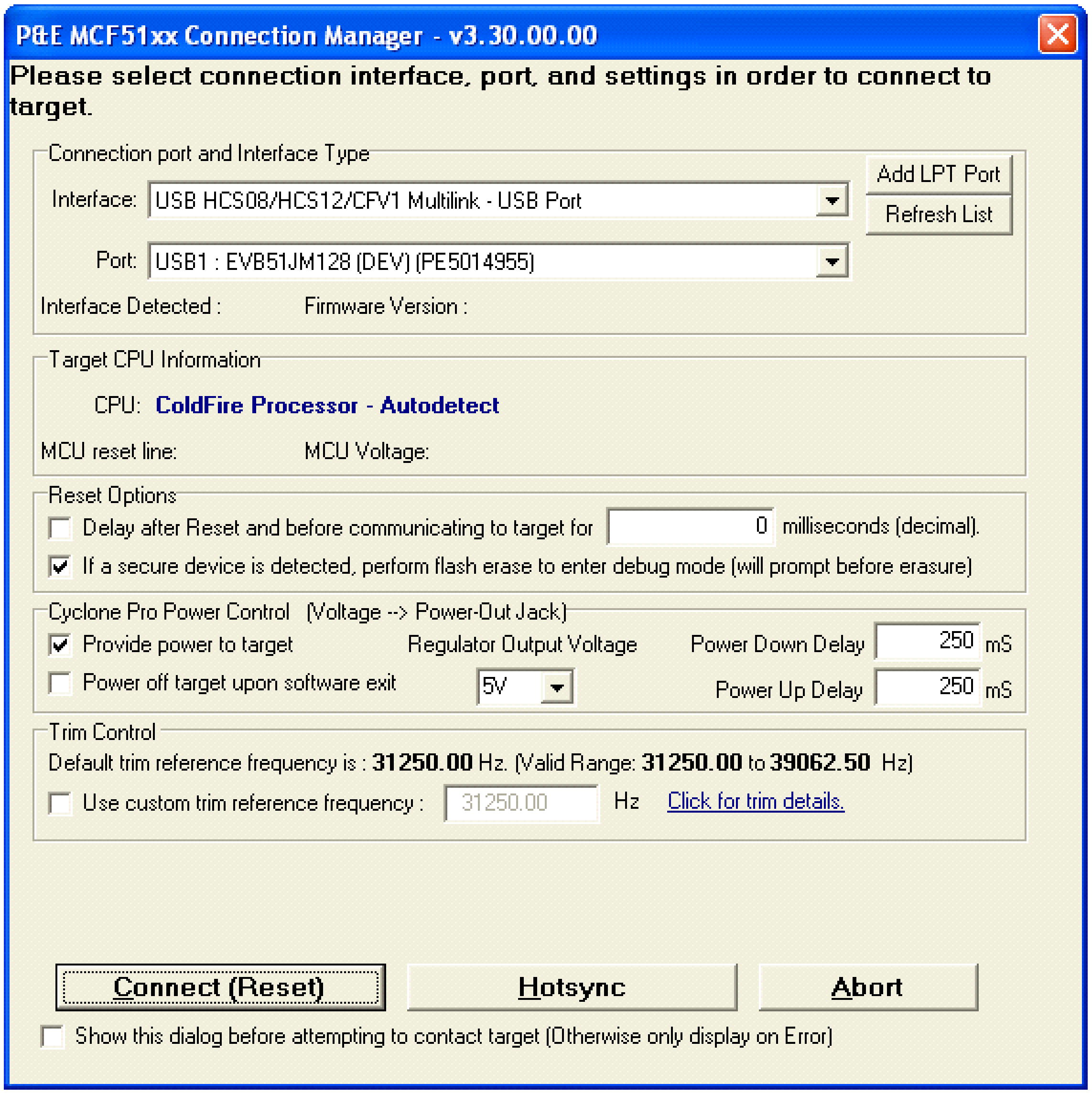
1. To run the application, click the button as shown in Figure A-4.



File	Code	Data		
Linker Files	0	0	•	
MQX Libraries	95204	7971	•	
USB Host Libraries	43342	1010	•	•
Source	2010	1093	•	•
usb_classes.h	0	0	•	
hidmouse.h	0	0	•	
hidmouse.c	2010	1093	•	•
12 files	137K	10074		

Figure A-4. Running the Application

2. Click **Connect (Reset)** button to connect to the hardware, as shown in Figure A-5.


P&E MCF51xx Connection Manager - v3.30.00.00

Please select connection interface, port, and settings in order to connect to target.

Connection port and Interface Type

Interface: USB HCS08/HCS12/CFV1 Multilink - USB Port

Port: USB1 : EVB51JM128 (DEV) (PE5014955)

Interface Detected : Firmware Version :

Target CPU Information

CPU: ColdFire Processor - Autodetect

MCU reset line: MCU Voltage:

Reset Options

☐ Delay after Reset and before communicating to target for 0 milliseconds (decimal).

☑ If a secure device is detected, perform flash erase to enter debug mode (will prompt before erasure)

Cyclone Pro Power Control (Voltage --> Power-Out Jack)

☑ Provide power to target — Regulator Output Voltage: 5V — Power Down Delay 250 mS

☐ Power off target upon software exit — Power Up Delay 250 mS

Trim Control

Default trim reference frequency is : **31250.00** Hz. (Valid Range: **31250.00** to **39062.50** Hz)

☐ Use custom trim reference frequency : 31250.00 Hz Click for trim details.

☐ Show this dialog before attempting to contact target (Otherwise only display on Error)

Figure A-5. Connection Manager

3. The pop-up in Figure A-6 opens to erase and program the built image to JM128 Flash.

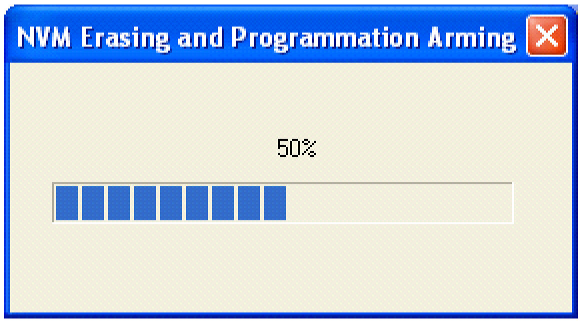


Figure A-6. Erasing and Programming window

4. The pop-up in Figure A-7 opens to load the built image to JM128 Flash.

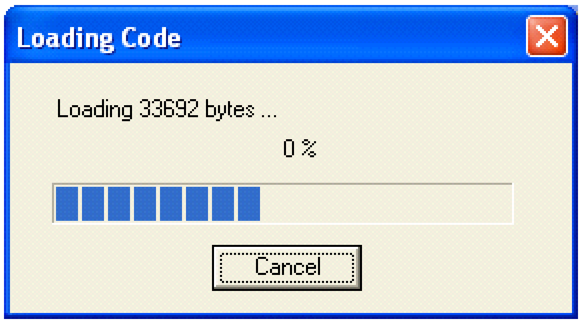
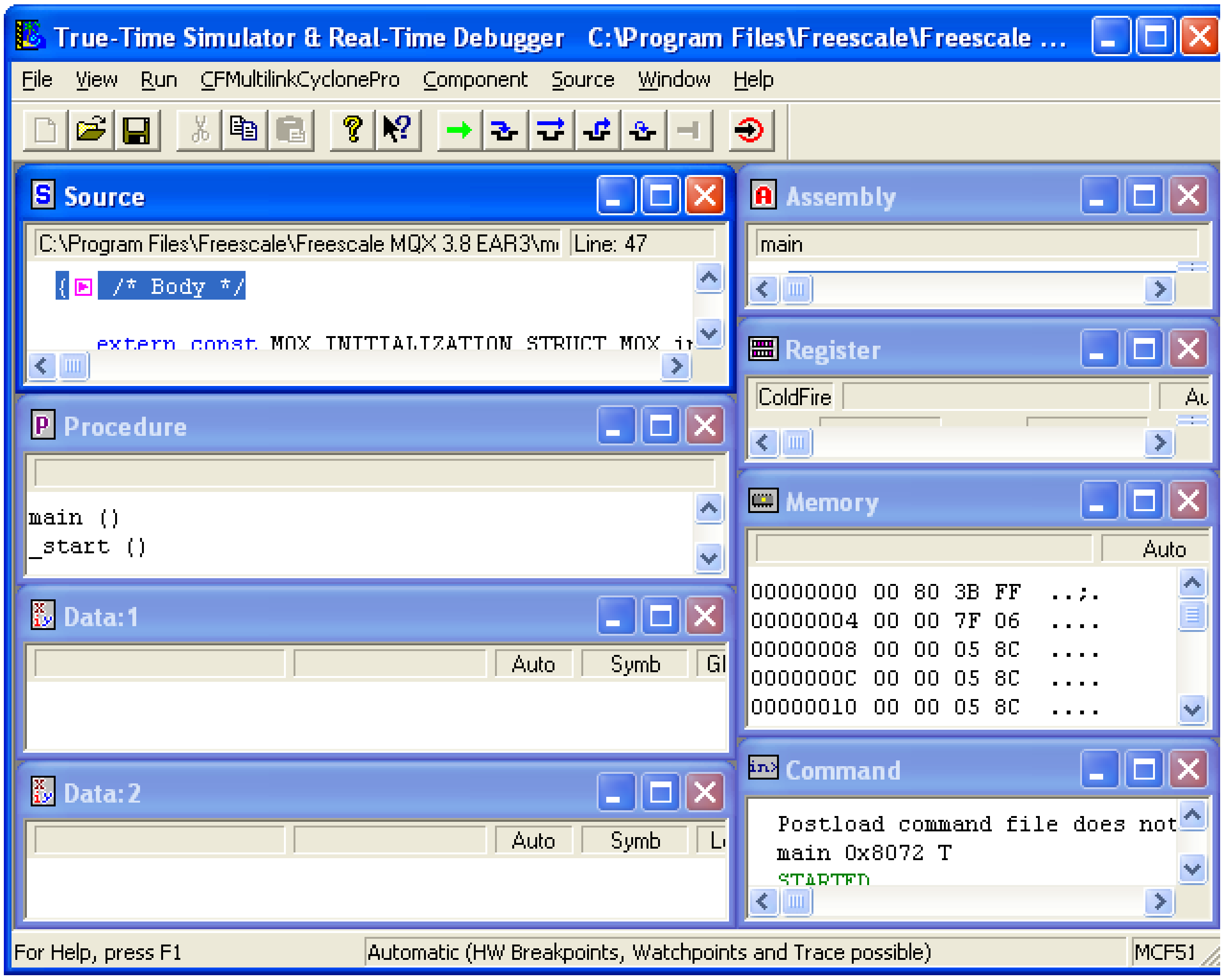


Figure A-7. Loading window

5. After the built image is loaded in the flash, the debugger window opens, as shown in Figure A-8. Click on the green arrow, as shown in Figure A-8, to run the programmed image.


True-Time Simulator & Real-Time Debugger C:\Program Files\Freescale\Freescale ...
File View Run CFMultilinkCyclonePro Component Source Window Help
Source
Assembly
Register
Procedure
Memory
Data:1
Data:2
Command
Auto
Symb

C:\Program Files\Freescale\Freescale MQX 3.8 EAR3\m Line: 47

```
{ /* Body */

extern const MQX_INITIALIZATION_STRUCT MQX_i
```

```
main
```

```
ColdFire
```

```
main ()
_start ()
```

```
00000000  00 80 3B FF   ..;.
00000004  00 00 7F 06   ....
00000008  00 00 05 8C   ....
0000000C  00 00 05 8C   ....
00000010  00 00 05 8C   ....
```

```
Postload command file does not
main 0x8072 T
STARTED
```

For Help, press F1 | Automatic (HW Breakpoints, Watchpoints and Trace possible) | MCF51

Figure A-8. Simulation and Realtime Debugger

A.2 Setup HyperTerminal to get log

To ensure that applications run correctly, the HyperTerminal is used to get events from the devices which connect to the CFV1. The following steps are used to configure HyperTerminal:

1. Open HyperTerminal applications as shown in Figure A-9.

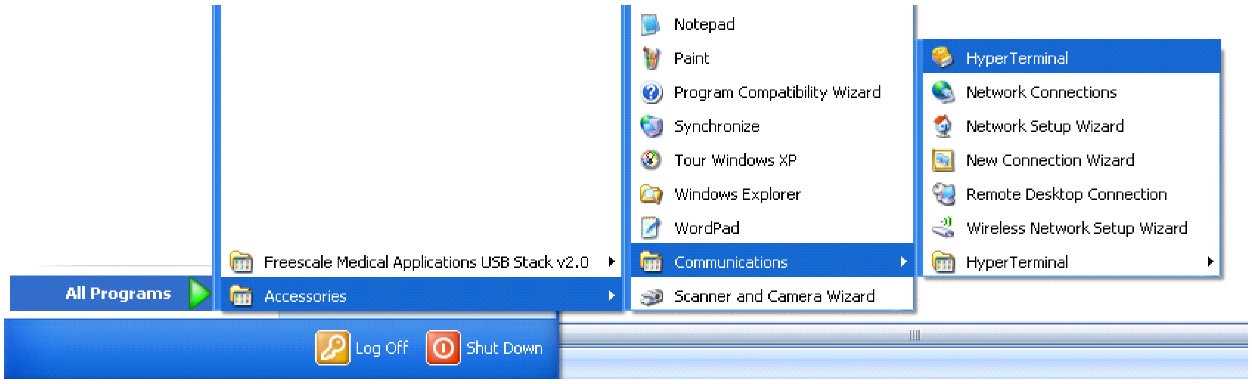
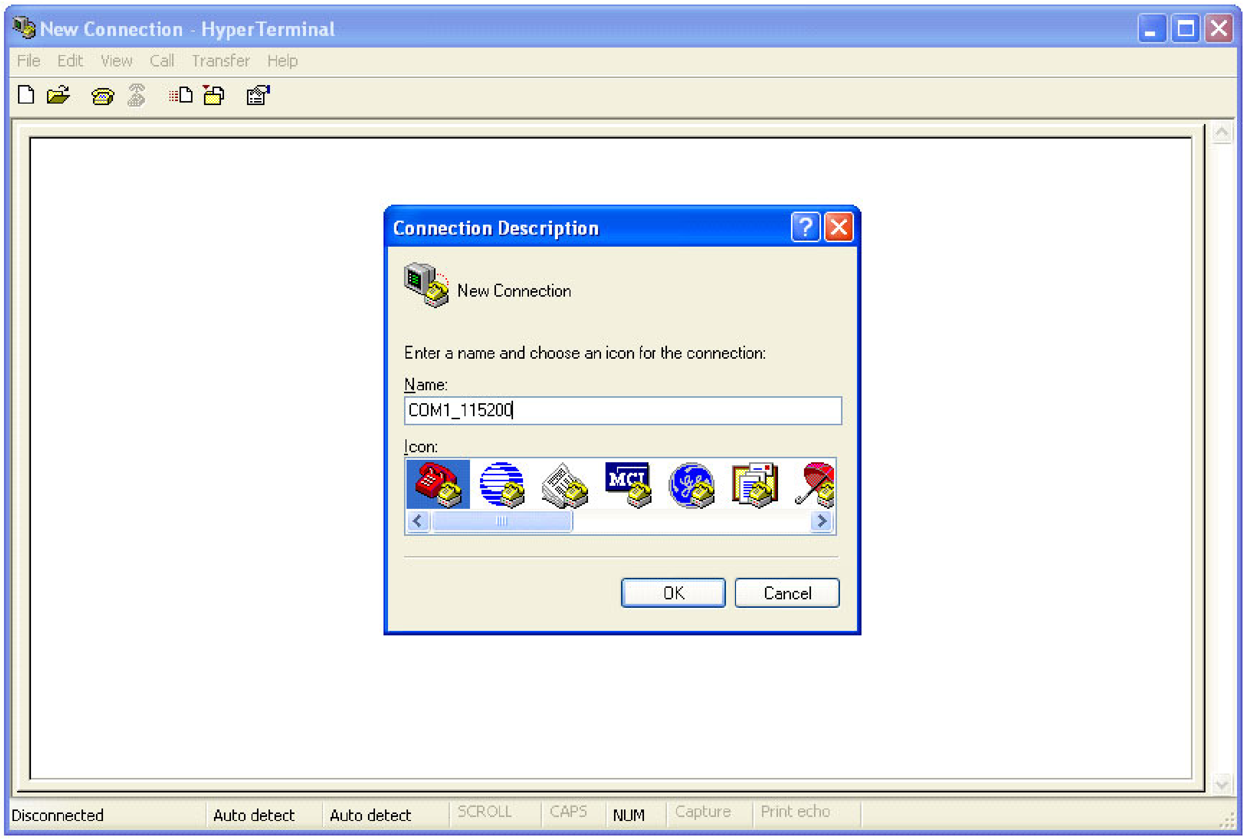


Figure A-9. Launch HyperTerminal Application

2. The HyperTerminal opens, as shown in Figure A-10. Enter the name of the connection and click on the **OK** button.



New Connection

Enter a name and choose an icon for the connection:

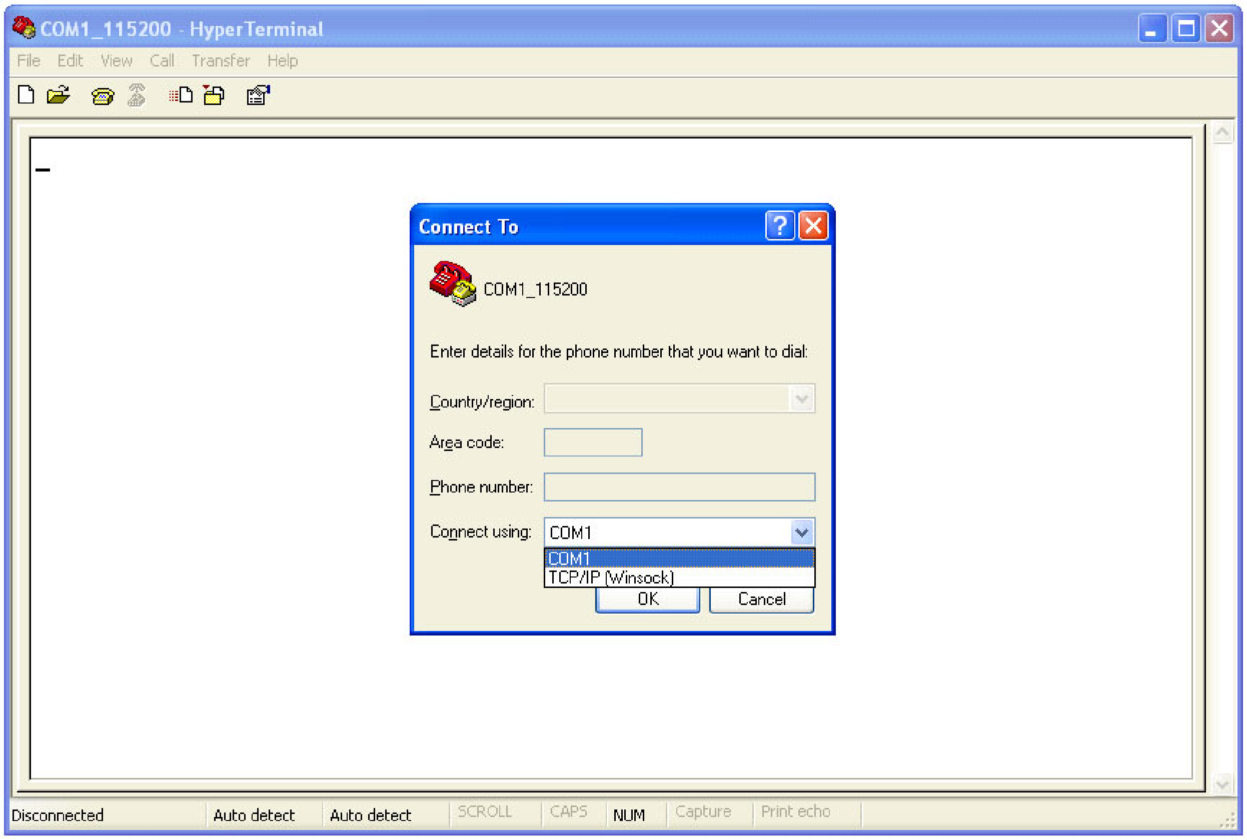
Name:

COM1_115200

Icon:

Figure A-10. HyperTerminal GUI

3. The window appears as shown in the Figure A-11. Select the COM port identical to the one that shows up on the device manager.



COM1_115200

Enter details for the phone number that you want to dial:

Country/region:

Area code:

Phone number:

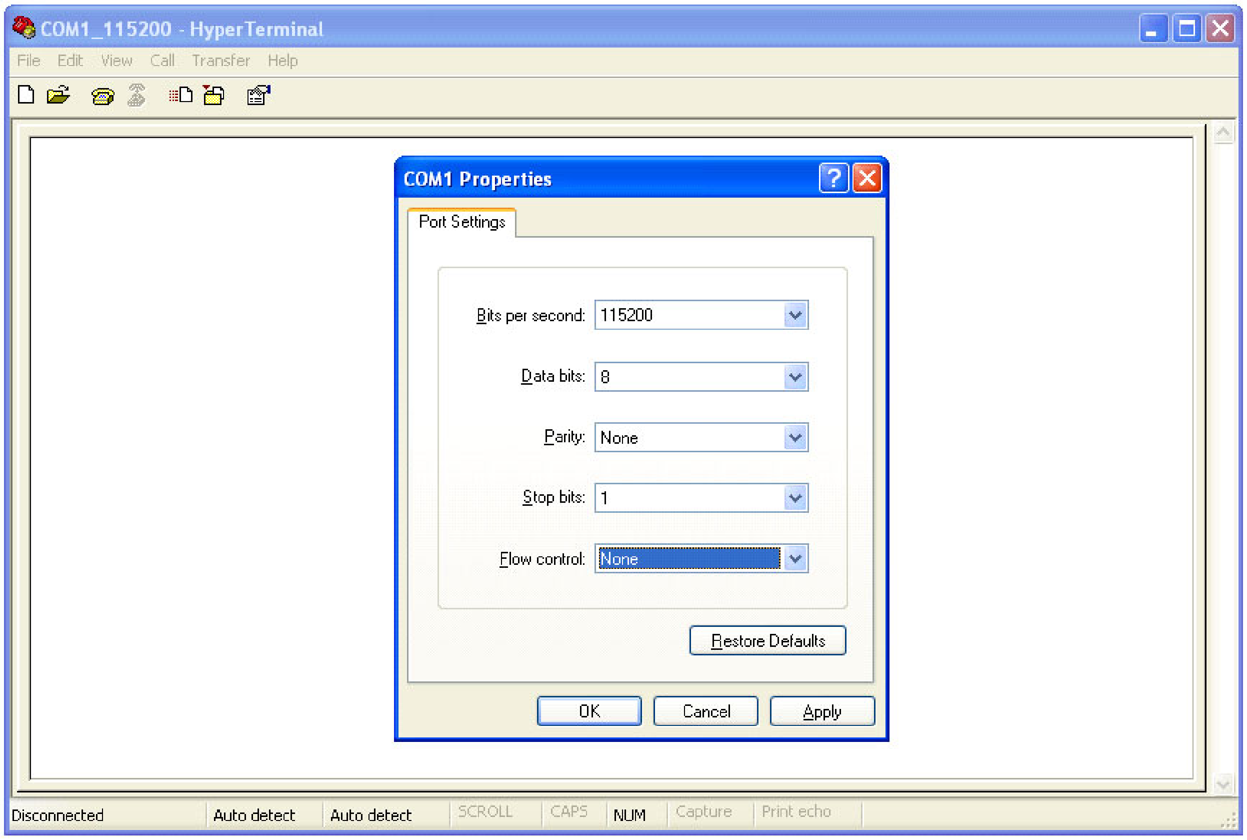
Connect using: COM1

COM1

TCP/IP (Winsock)

Figure A-11. Connect using COM1

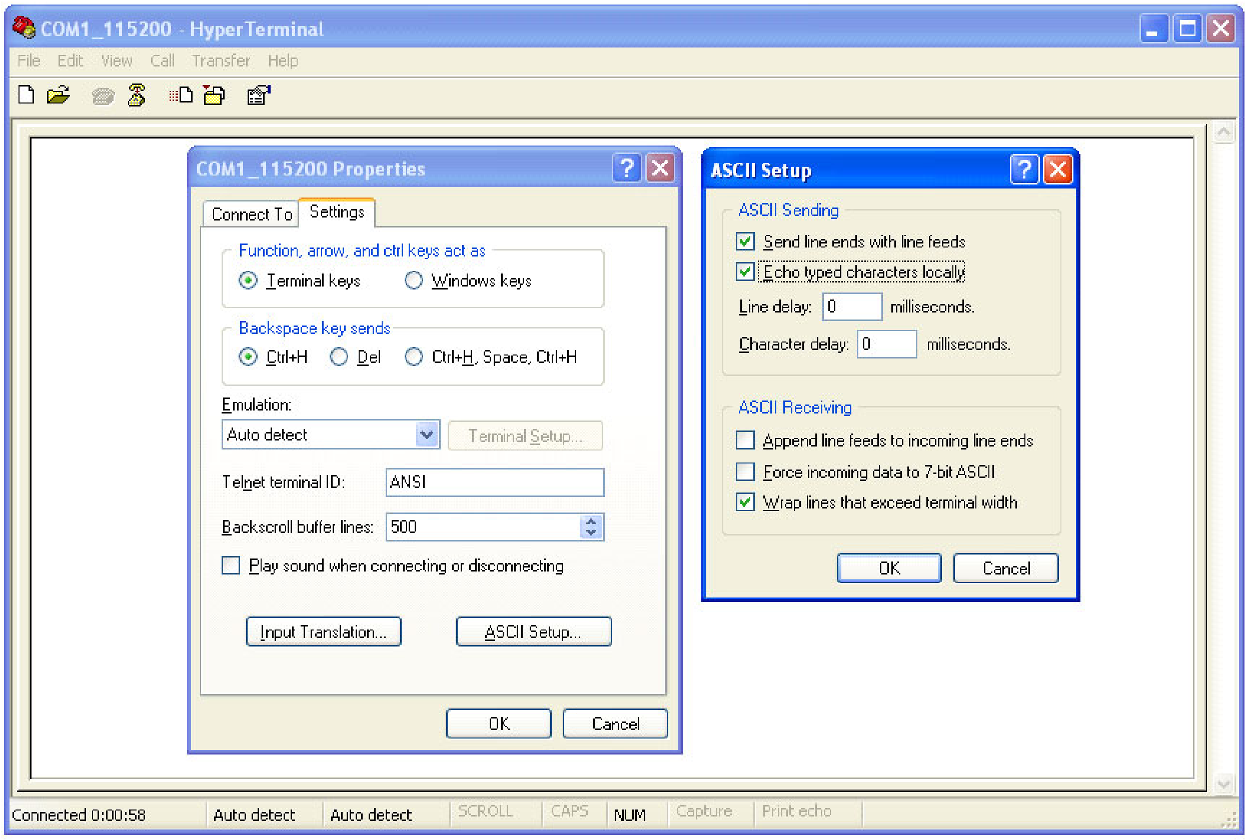
4. Configure the virtual COM port baudrate and other properties as shown in Figure A-12.



Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Figure A-12. COM1 Properties

5. Configure the HyperTerminal as shown in Figure A-13. Click on the **OK** button to submit changes.



Function, arrow, and ctrl keys act as: [✓ Terminal keys] Windows keys

Backspace key sends: [✓ Ctrl+H] Del Ctrl+H, Space, Ctrl+H

Emulation: Auto detect (Terminal Setup...)

Telnet terminal ID: ANSI

Backscroll buffer lines: 500

☐ Play sound when connecting or disconnecting

ASCII Sending

☑ Send line ends with line feeds

☑ Echo typed characters locally

Line delay: 0 milliseconds.

Character delay: 0 milliseconds.

ASCII Receiving

☐ Append line feeds to incoming line ends

☐ Force incoming data to 7-bit ASCII

☑ Wrap lines that exceed terminal width

Figure A-13. Configure COM1_115200 - HyperTerminal

6. The HyperTerminal is configured now.

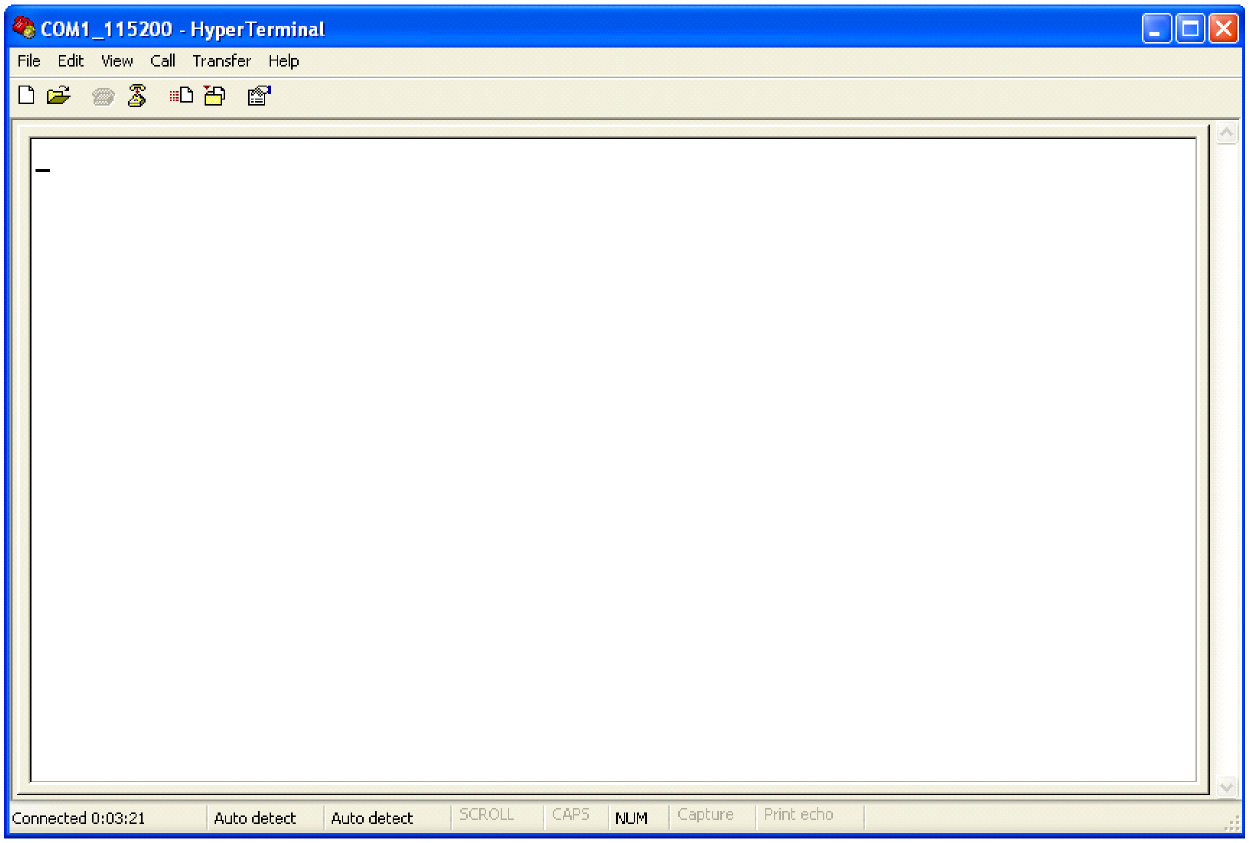


Figure A-14. COM1_115200 is configured

Appendix B Human Interface Device (HID) Demo

B.1 Setting Up the Demo

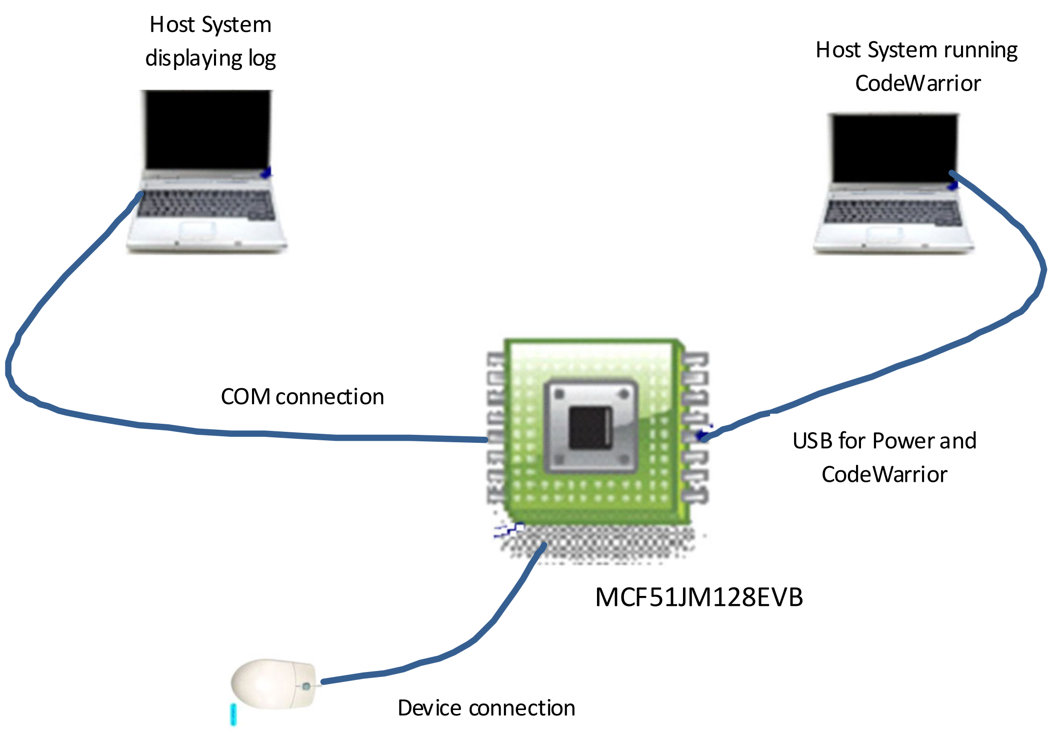


Figure B-1. HID Demo Setup

Figure B-1 describes the HID demo setup. The MCF51JM128EVB is used as the USB Host. The MCF51JM128EVB board is connected to the first personal computer by using USB cables. This computer is used to supply power to the board and is used to program the image to the flash. The MCF51JM128EVB board is connected to the second personal computer via COM port. This computer is used to log events happening in the USB Host. The device (mouse or keyboard) is connected to the MCF51JM128EVBboard. Although Figure B-1 shows two computers, the connection can also be achieved using one computer.

B.2 Running the Demo

The HID projects are located in:

<MQX installation folder>\usb\host\examples\hid\mouse\

There are 3 applications of HID classes.

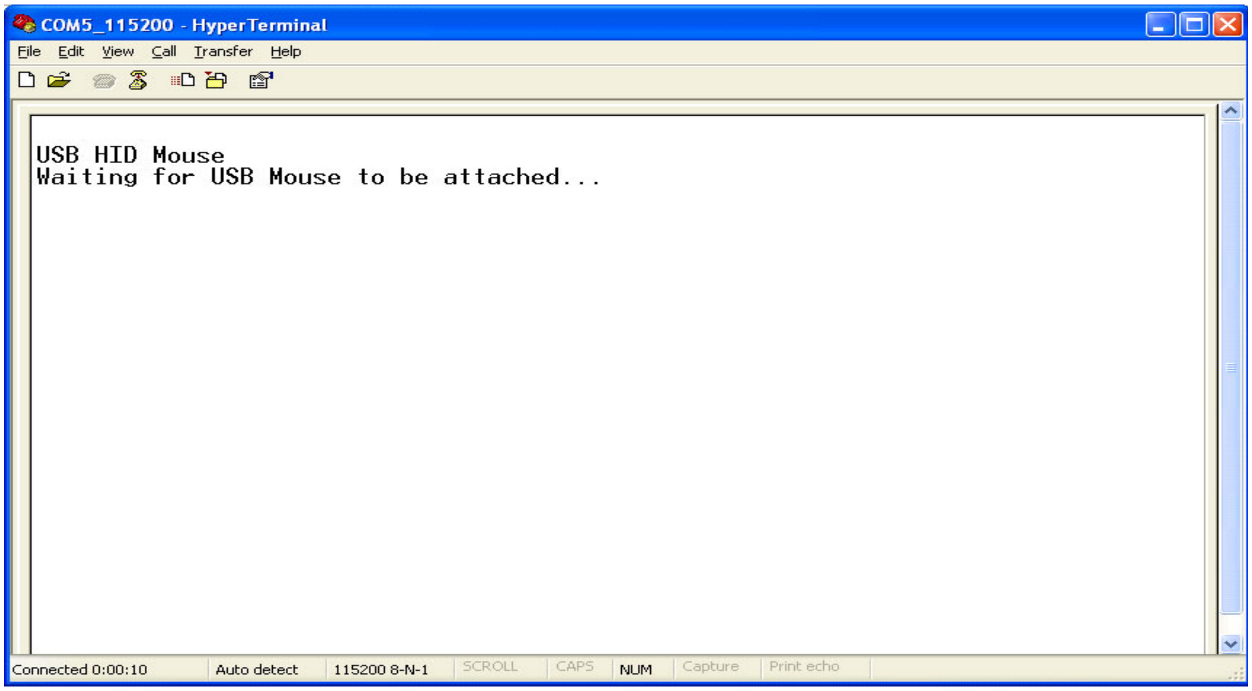
- Mouse
- Keyboard

- Keyboard and Mouse

B.2.1 Mouse demo

Follow the steps below to run the mouse demo:

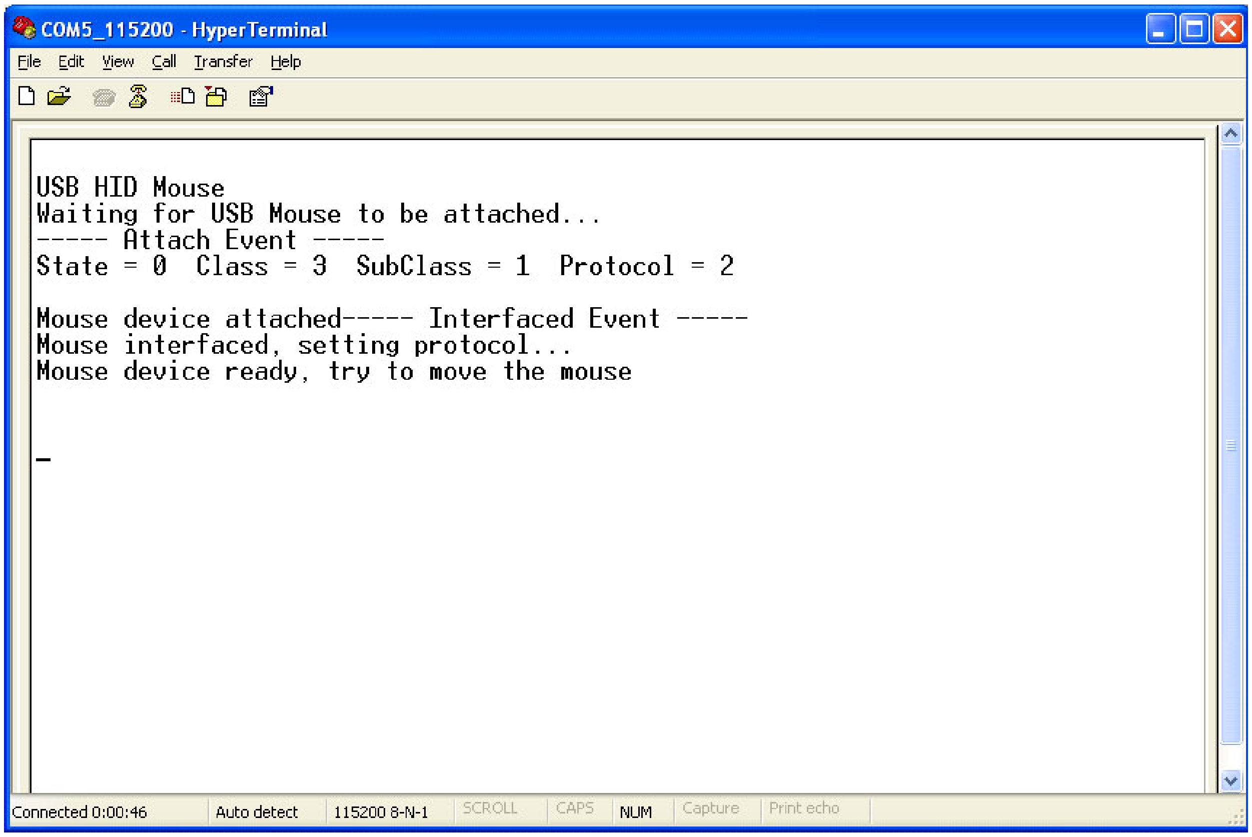
1. Open and load image of Mouse applications to board.
2. After the image has been loaded successfully, the Hyperterminal appears as shown in the Figure B-2:


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
USB HID Mouse
Waiting for USB Mouse to be attached...
```

Figure B-2. The USB Host is waiting the mouse attachment event

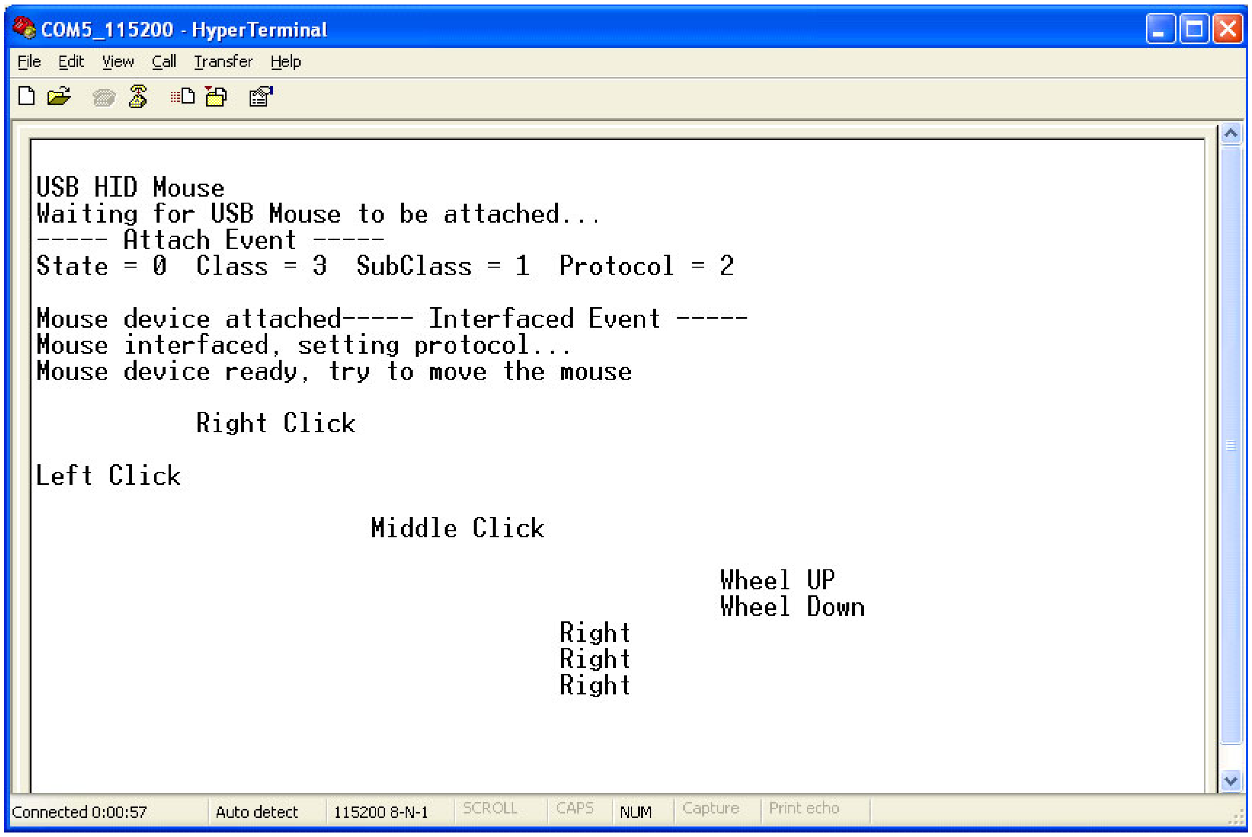
3. Plug-in mouse to the board. The Hyperterminal appears as shown in Figure B-3:


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
USB HID Mouse
Waiting for USB Mouse to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 2

Mouse device attached----- Interfaced Event -----
Mouse interfaced, setting protocol...
Mouse device ready, try to move the mouse


_
```

Figure B-3. Mouse attached

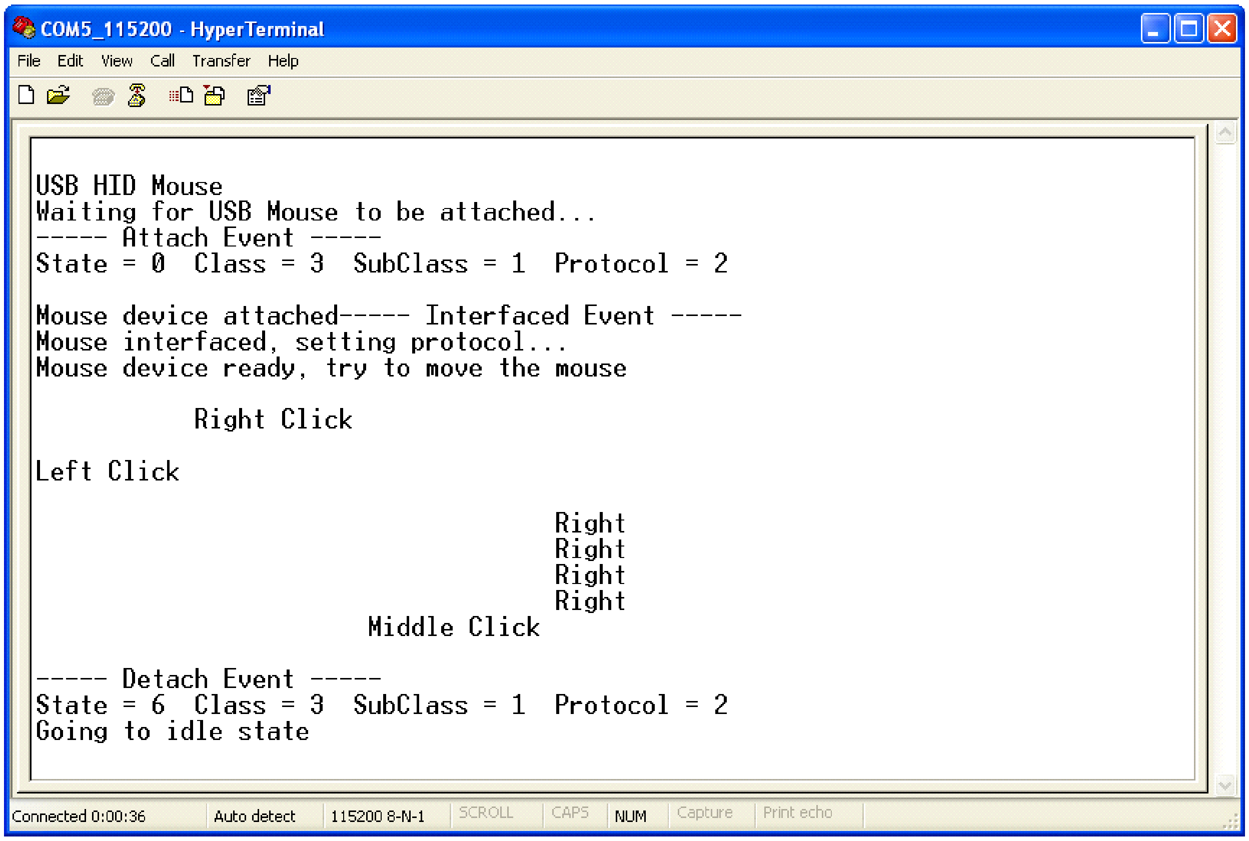
4. When events are implemented (click right mouse, click left mouse ...), they are registered as shown in Figure B-4:



```
USB HID Mouse
Waiting for USB Mouse to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 2

Mouse device attached----- Interfaced Event -----
Mouse interfaced, setting protocol...
Mouse device ready, try to move the mouse

           Right Click

Left Click

                       Middle Click

                                               Wheel UP
                                               Wheel Down
                                    Right
                                    Right
                                    Right
```

Figure B-4. Events from mouse

5. Unplug mouse from the board. The Hyperterminal displays a message as shown in Figure B-5:



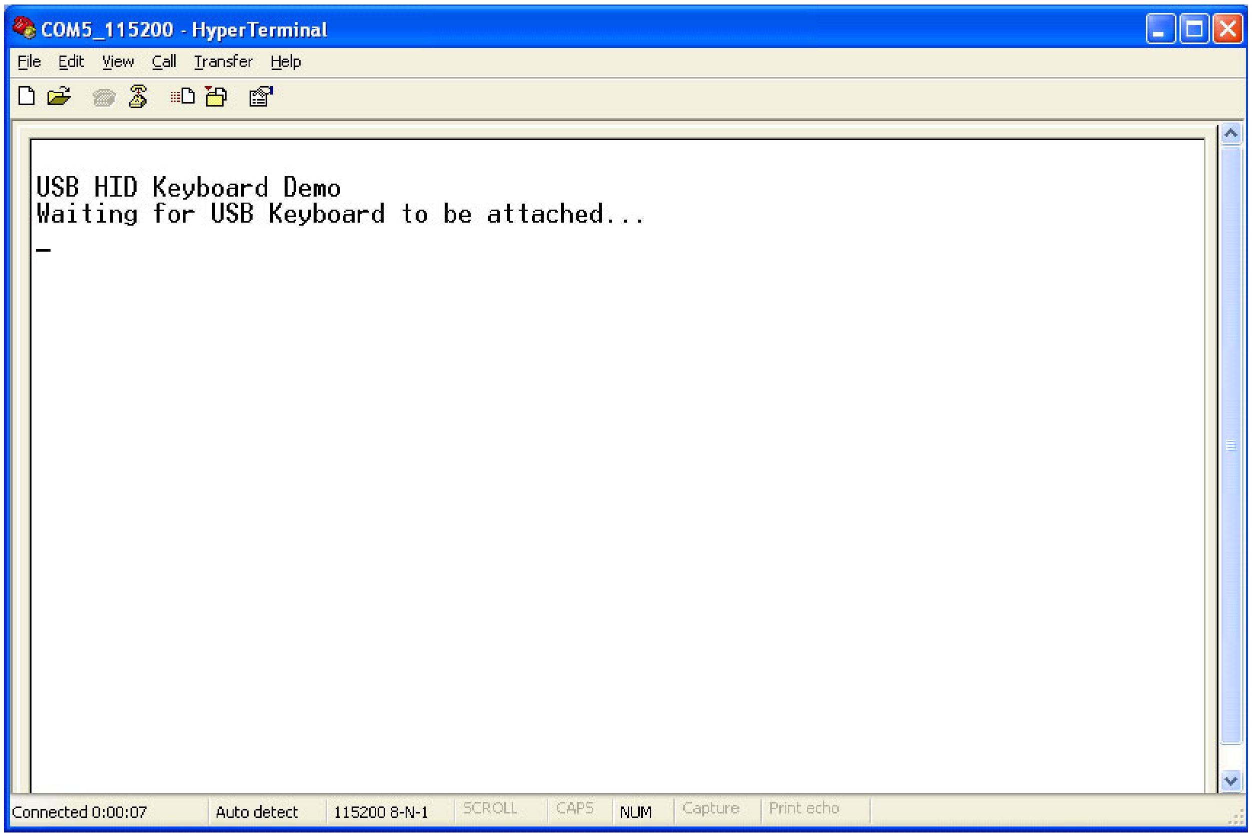
```
USB HID Mouse
Waiting for USB Mouse to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 2

Mouse device attached----- Interfaced Event -----
Mouse interfaced, setting protocol...
Mouse device ready, try to move the mouse

           Right Click

Left Click

                                    Right
                                    Right
                                    Right
                                    Right
                       Middle Click

----- Detach Event -----
State = 6  Class = 3  SubClass = 1  Protocol = 2
Going to idle state
```

Figure B-5. Mouse detached

B.2.2 Keyboard demo

Follow these steps below to run the keyboard demo:

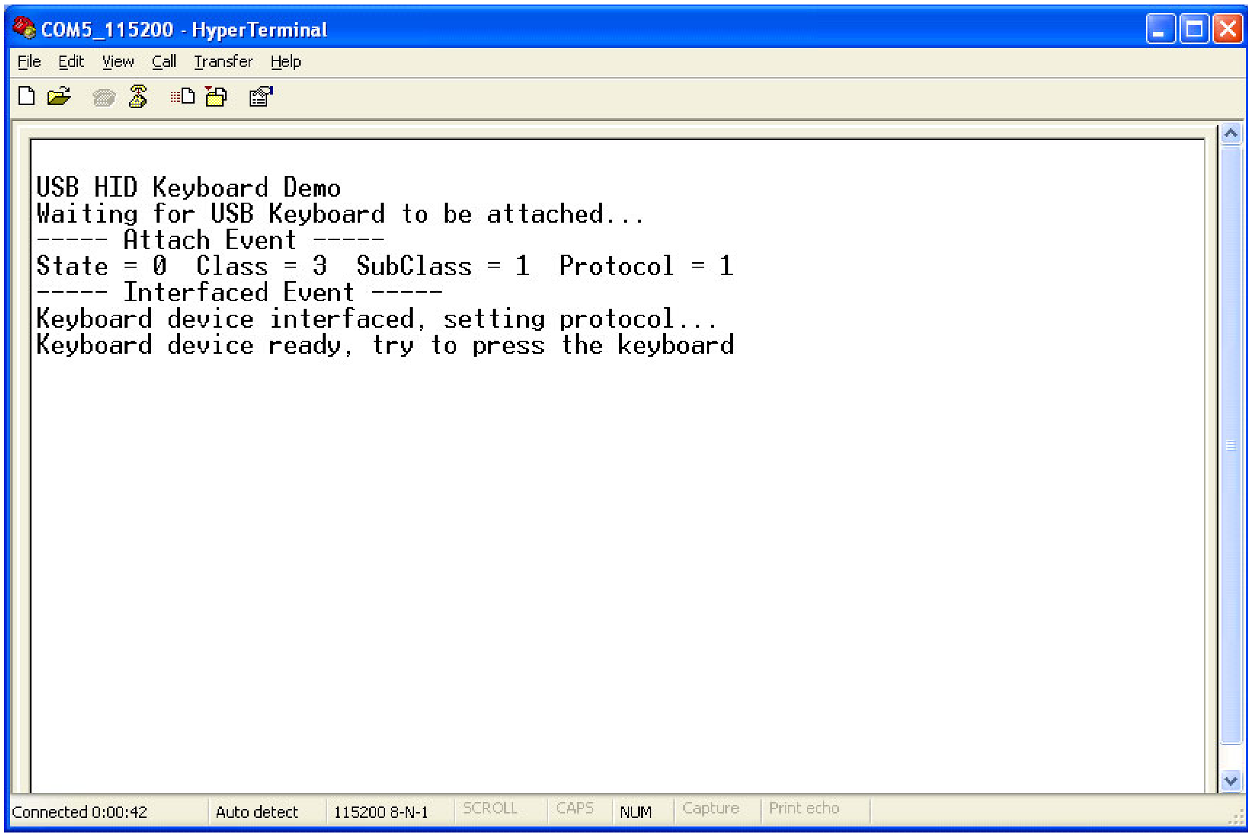
1. Open and load image to the board.
2. Run the demo. First, the USB Host waits for the device attachment event. The HyperTerminal displays a message as shown in Figure B-6:


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
USB HID Keyboard Demo
Waiting for USB Keyboard to be attached...
_
```

Figure B-6. The USB Host is waiting the keyboard attachment event

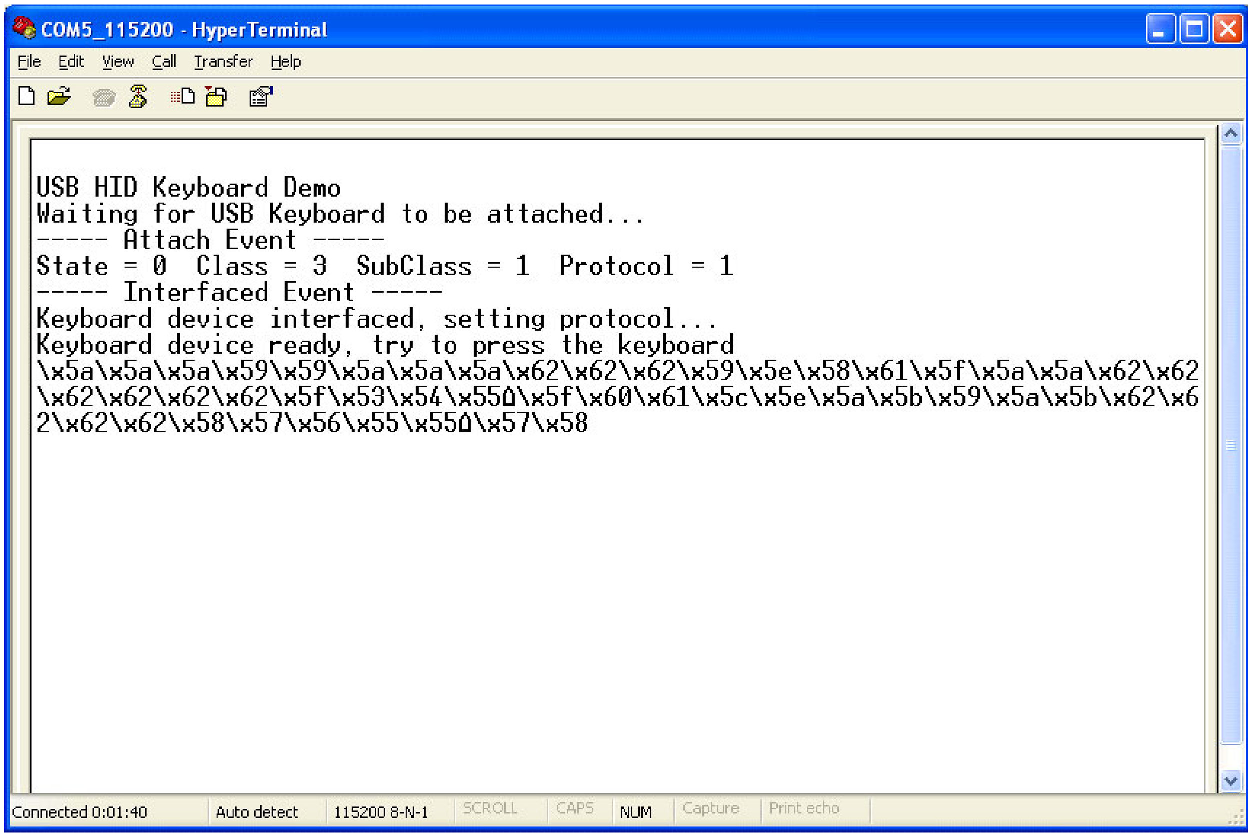
3. Plug-in the Keyboard. The Hyperterminal displays a message as shown in Figure B-7:


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
USB HID Keyboard Demo
Waiting for USB Keyboard to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 1
----- Interfaced Event -----
Keyboard device interfaced, setting protocol...
Keyboard device ready, try to press the keyboard
```

Figure B-7. Keyboard attached

4. Type some characters. The Hexa format of these characters will be displayed in HyperTerminal as shown in Figure B-8:


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
USB HID Keyboard Demo
Waiting for USB Keyboard to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 1
----- Interfaced Event -----
Keyboard device interfaced, setting protocol...
Keyboard device ready, try to press the keyboard
\x5a\x5a\x5a\x59\x59\x5a\x5a\x5a\x62\x62\x62\x59\x5e\x58\x61\x5f\x5a\x5a\x62\x62
\x62\x62\x62\x62\x5f\x53\x54\x55D\x5f\x60\x61\x5c\x5e\x5a\x5b\x59\x5a\x5b\x62\x6
2\x62\x62\x58\x57\x56\x55\x55D\x57\x58
```

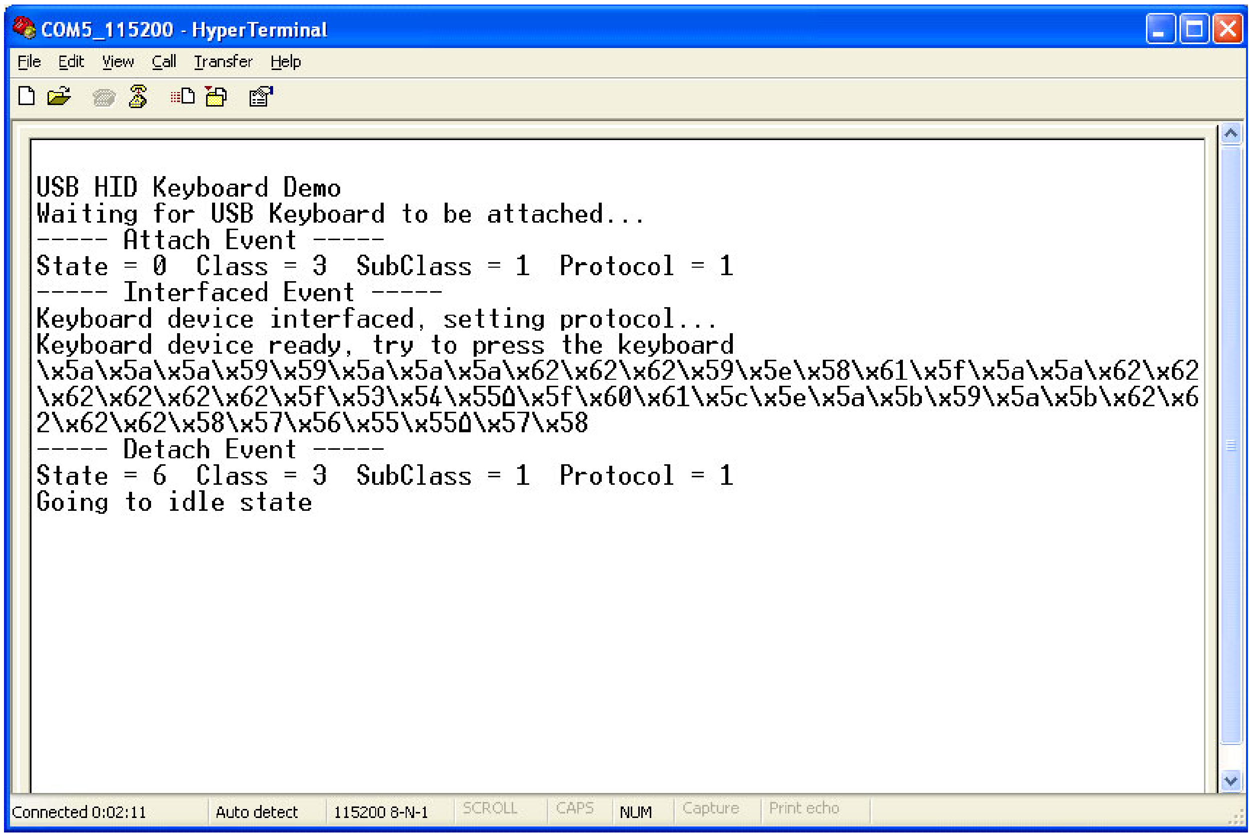
```
Connected 0:01:40    Auto detect    115200 8-N-1    SCROLL    CAPS    NUM    Capture    Print echo
```

Figure B-8. Typing character from keyboard

NOTE

The Hyperterminal only shows the ASCII code in Hexa of each character.

5. Unplug the keyboard. The Hyperterminal displays a message as shown in Figure B-9:



```
USB HID Keyboard Demo
Waiting for USB Keyboard to be attached...
----- Attach Event -----
State = 0  Class = 3  SubClass = 1  Protocol = 1
----- Interfaced Event -----
Keyboard device interfaced, setting protocol...
Keyboard device ready, try to press the keyboard
\x5a\x5a\x5a\x59\x59\x5a\x5a\x5a\x62\x62\x62\x59\x5e\x58\x61\x5f\x5a\x5a\x62\x62
\x62\x62\x62\x62\x5f\x53\x54\x55Ω\x5f\x60\x61\x5c\x5e\x5a\x5b\x59\x5a\x5b\x62\x6
2\x62\x62\x58\x57\x56\x55\x55Ω\x57\x58
----- Detach Event -----
State = 6  Class = 3  SubClass = 1  Protocol = 1
Going to idle state
```

Figure B-9. Keyboard detached

B.2.3 Mouse and keyboard demo

This application is the combination of the two above. It supplies a convenient choice for users to alternate the HID device (the mouse or the keyboard) they want to work with.

Appendix C Mass Storage Device (MSD) Demo

C.1 Setting Up the Demo

Set the system up as described in the Appendix B, "Human Interface Device (HID) Demo."

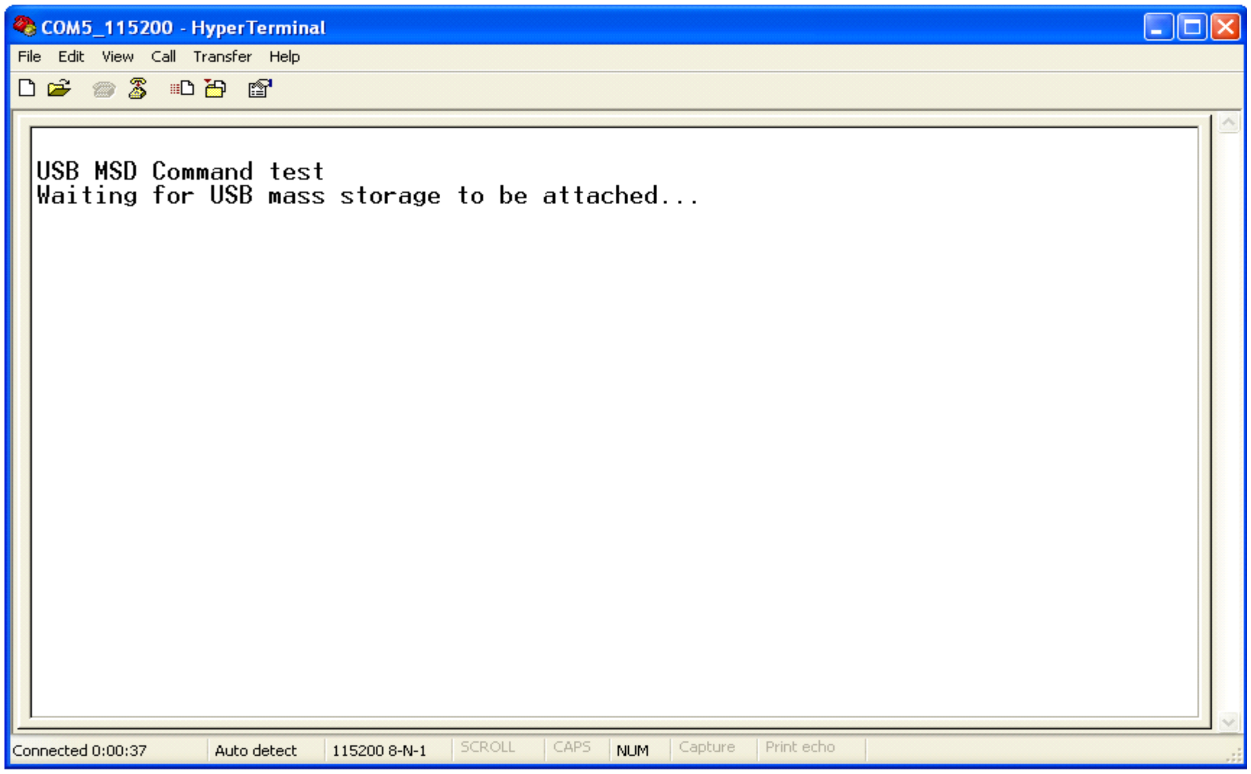
C.2 Running the Demo

To run this Demo, follow the steps below:

1. Open the MSD Demo project and load the image to the board.

<install_dir>\usb\host\examples\msd\msd_commands

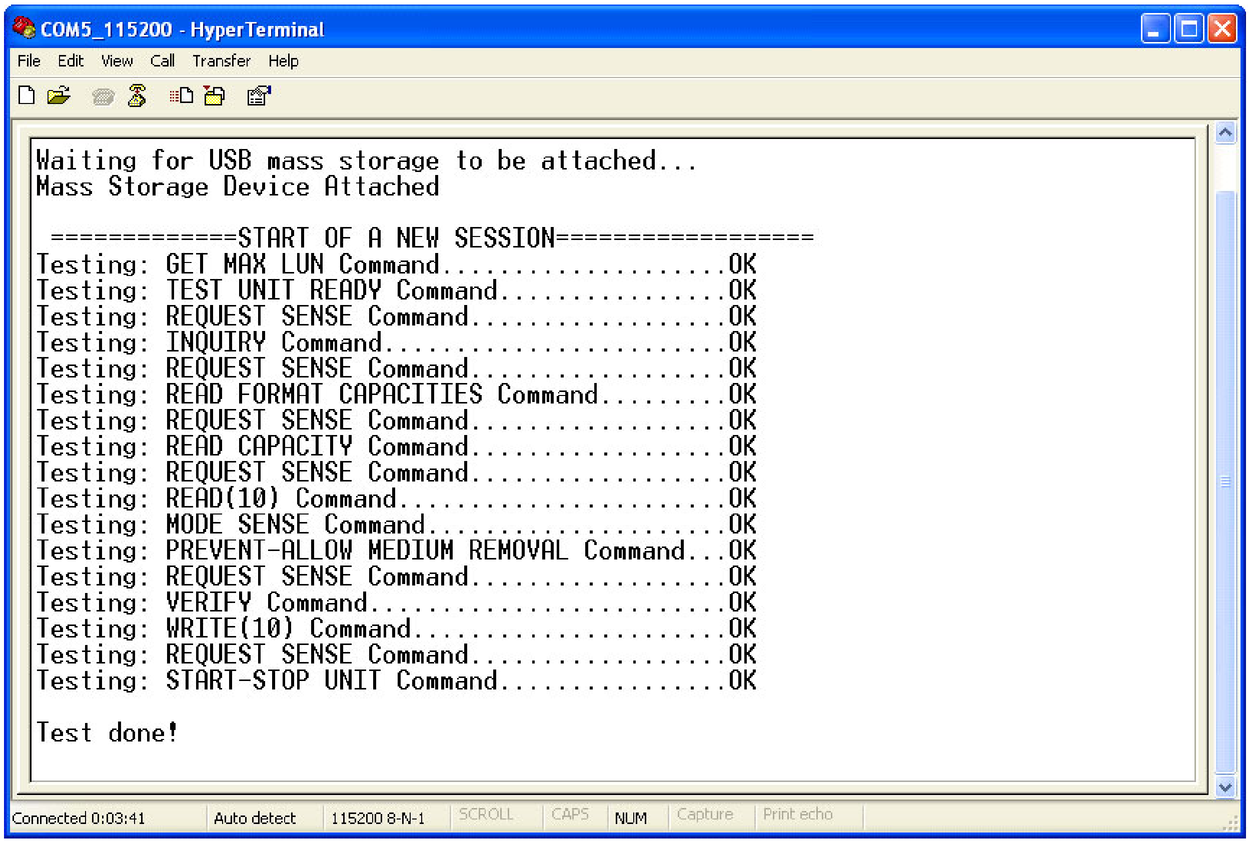
2. Run the Demo. The HyperTerminal displays a message as shown in Figure C-1.



```
USB MSD Command test
Waiting for USB mass storage to be attached...
```

Figure C-1. USB Host waits USB mass storage to be attached

3. Attach the USB Mass Storage to this board. The HyperTerminal displays a test result as shown in Figure C-2.

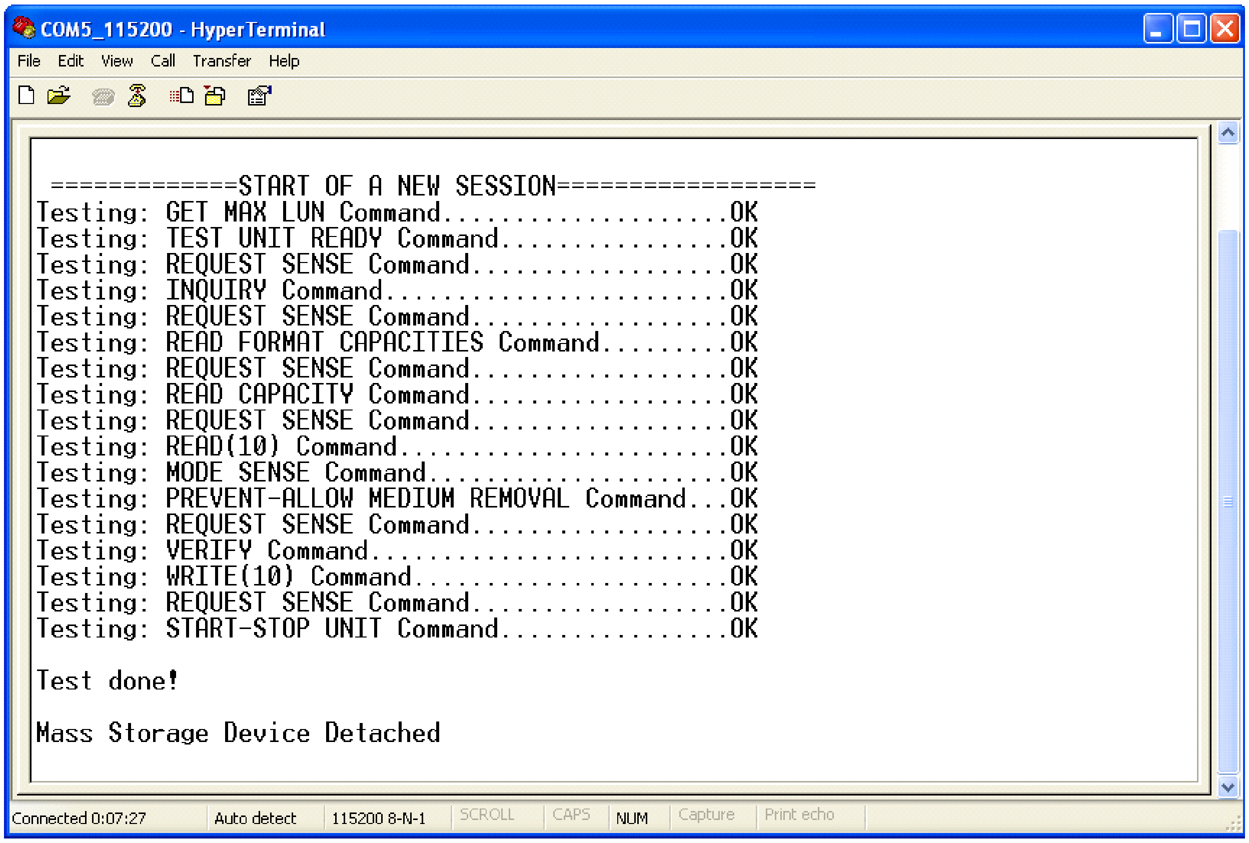

COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
Waiting for USB mass storage to be attached...
Mass Storage Device Attached

 =============START OF A NEW SESSION=================
Testing: GET MAX LUN Command...................OK
Testing: TEST UNIT READY Command...............OK
Testing: REQUEST SENSE Command.................OK
Testing: INQUIRY Command.......................OK
Testing: REQUEST SENSE Command.................OK
Testing: READ FORMAT CAPACITIES Command........OK
Testing: REQUEST SENSE Command.................OK
Testing: READ CAPACITY Command.................OK
Testing: REQUEST SENSE Command.................OK
Testing: READ(10) Command......................OK
Testing: MODE SENSE Command....................OK
Testing: PREVENT-ALLOW MEDIUM REMOVAL Command...OK
Testing: REQUEST SENSE Command.................OK
Testing: VERIFY Command........................OK
Testing: WRITE(10) Command.....................OK
Testing: REQUEST SENSE Command.................OK
Testing: START-STOP UNIT Command...............OK

Test done!
```

Figure C-2. Test result

The message shows that all test cases have passed.

4. Unplug the device. The HyperTerminal displays the message that the device is detached as shown in Figure C-3.



```
 =============START OF A NEW SESSION==================
Testing: GET MAX LUN Command...................OK
Testing: TEST UNIT READY Command...............OK
Testing: REQUEST SENSE Command.................OK
Testing: INQUIRY Command.......................OK
Testing: REQUEST SENSE Command.................OK
Testing: READ FORMAT CAPACITIES Command........OK
Testing: REQUEST SENSE Command.................OK
Testing: READ CAPACITY Command.................OK
Testing: REQUEST SENSE Command.................OK
Testing: READ(10) Command......................OK
Testing: MODE SENSE Command....................OK
Testing: PREVENT-ALLOW MEDIUM REMOVAL Command...OK
Testing: REQUEST SENSE Command.................OK
Testing: VERIFY Command........................OK
Testing: WRITE(10) Command.....................OK
Testing: REQUEST SENSE Command.................OK
Testing: START-STOP UNIT Command...............OK

Test done!

Mass Storage Device Detached
```

Figure C-3. USB mass storage detached

Appendix D Virtual Communication (COM) Demo

The 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.

D.1 Setting Up the Demo

Set the system as described in the Appendix B "Human Interface Device (HID) Demo.

NOTE

To run this demo, a CDC device is necessary.

In this demo, the data entered from the keyboard is echoed and displayed in HyperTerminal.

The Data flow in the CDC Demo is shown in Figure D-1.

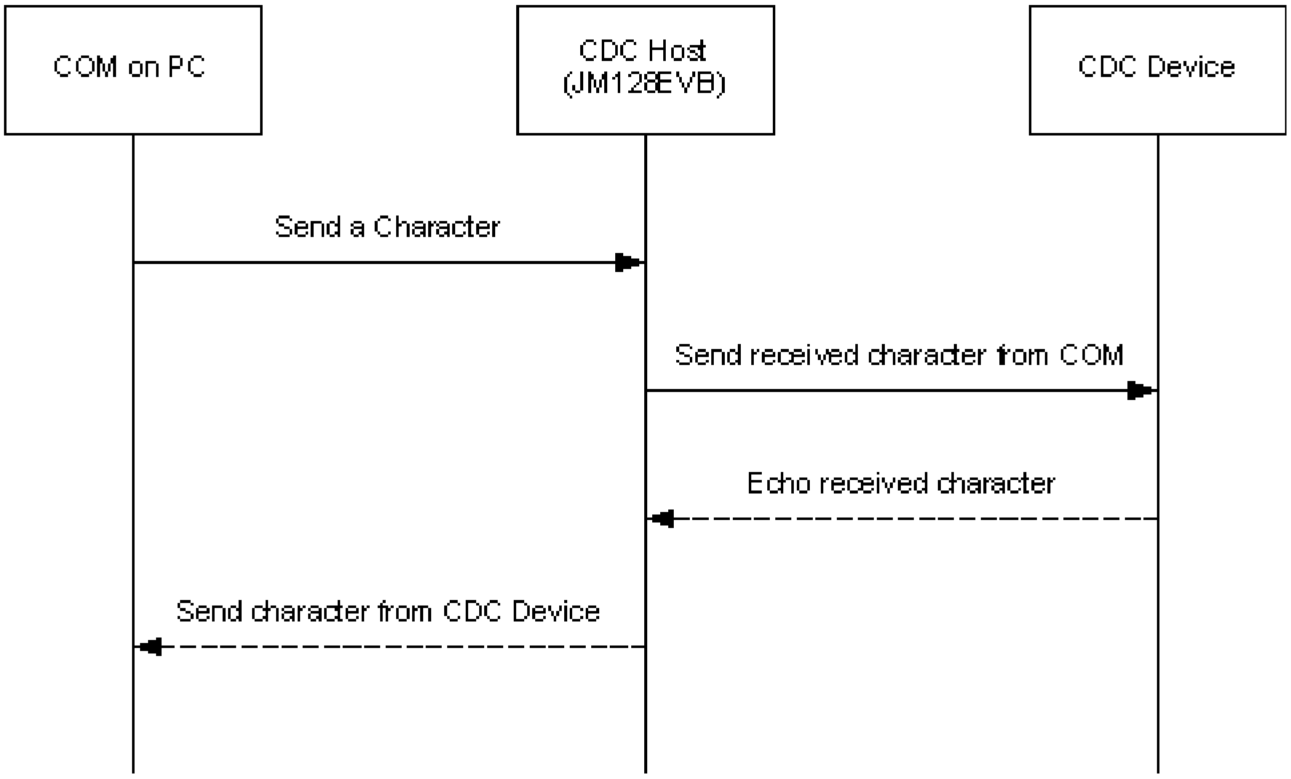


Figure D-1. Data flow

D.2 Running the Demo

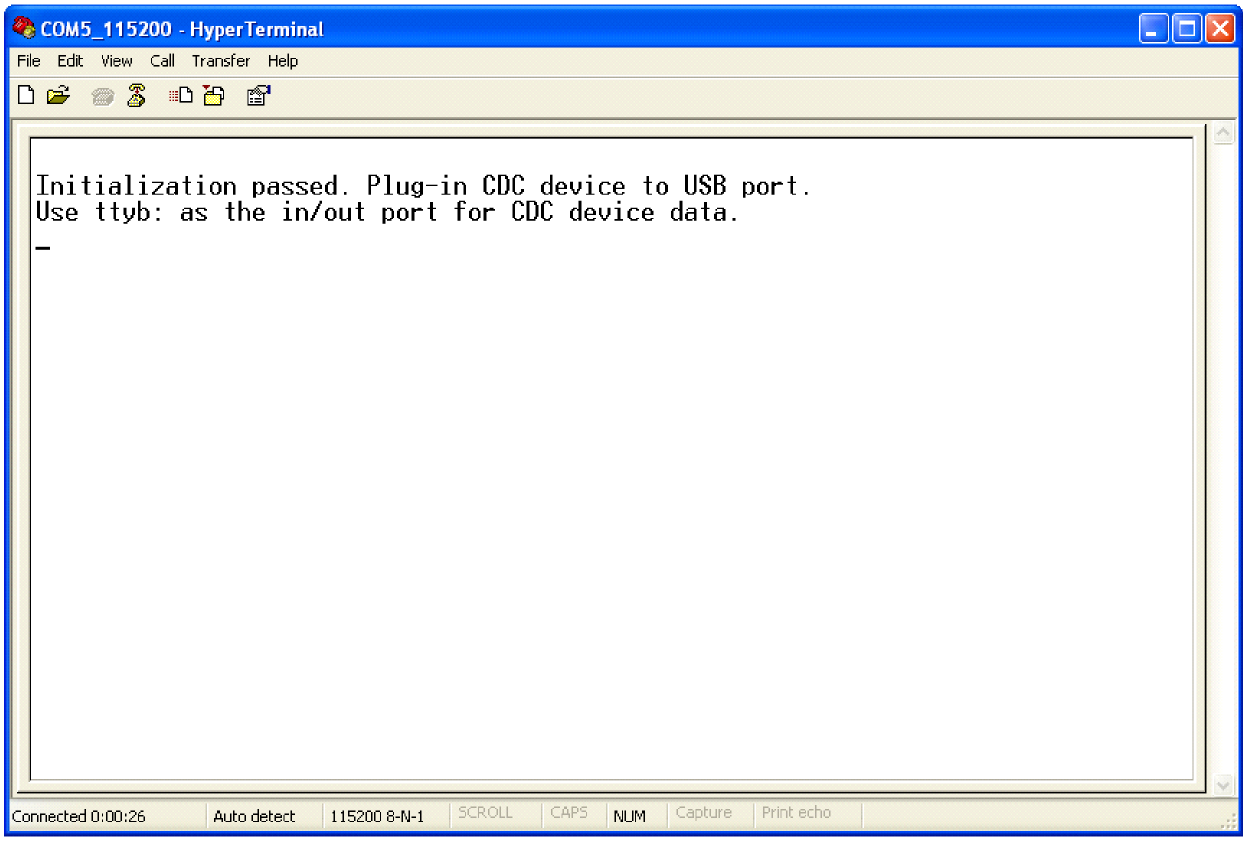
To run the CDC Demo, follow the steps below:

1. Open the CDC Demo project and load the image to the board.

The CDC application project is located in

<install_dir>\usb\host\examples\cdc\cdc_serial\

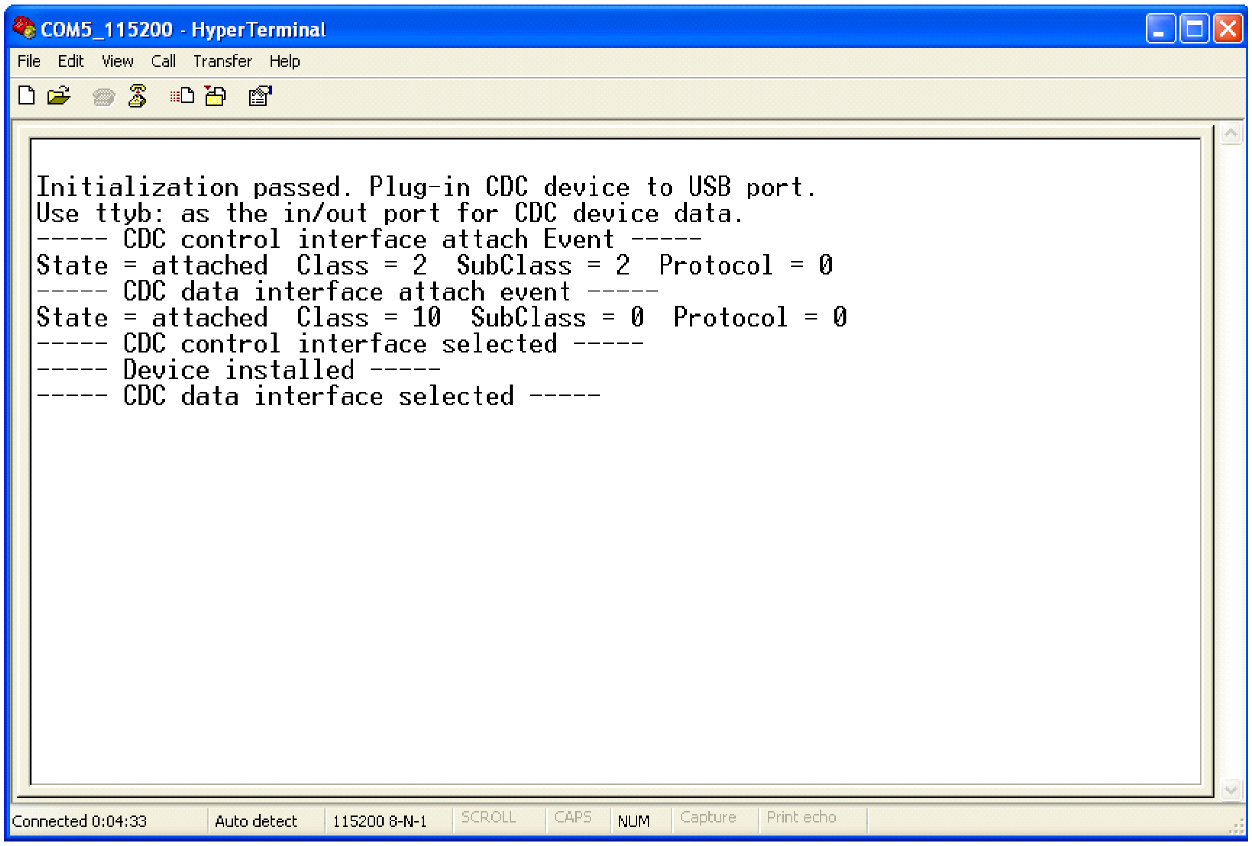
2. Connect COM1 of the board to the PC by using the steps shown in A.2 “Setup HyperTerminal to get log”.
3. Run the Demo. The HyperTerminal displays a message as shown in Figure D-2.



```
Initialization passed. Plug-in CDC device to USB port.
Use ttyb: as the in/out port for CDC device data.
_
```

Figure D-2. USB Host waits CDC device plug-in

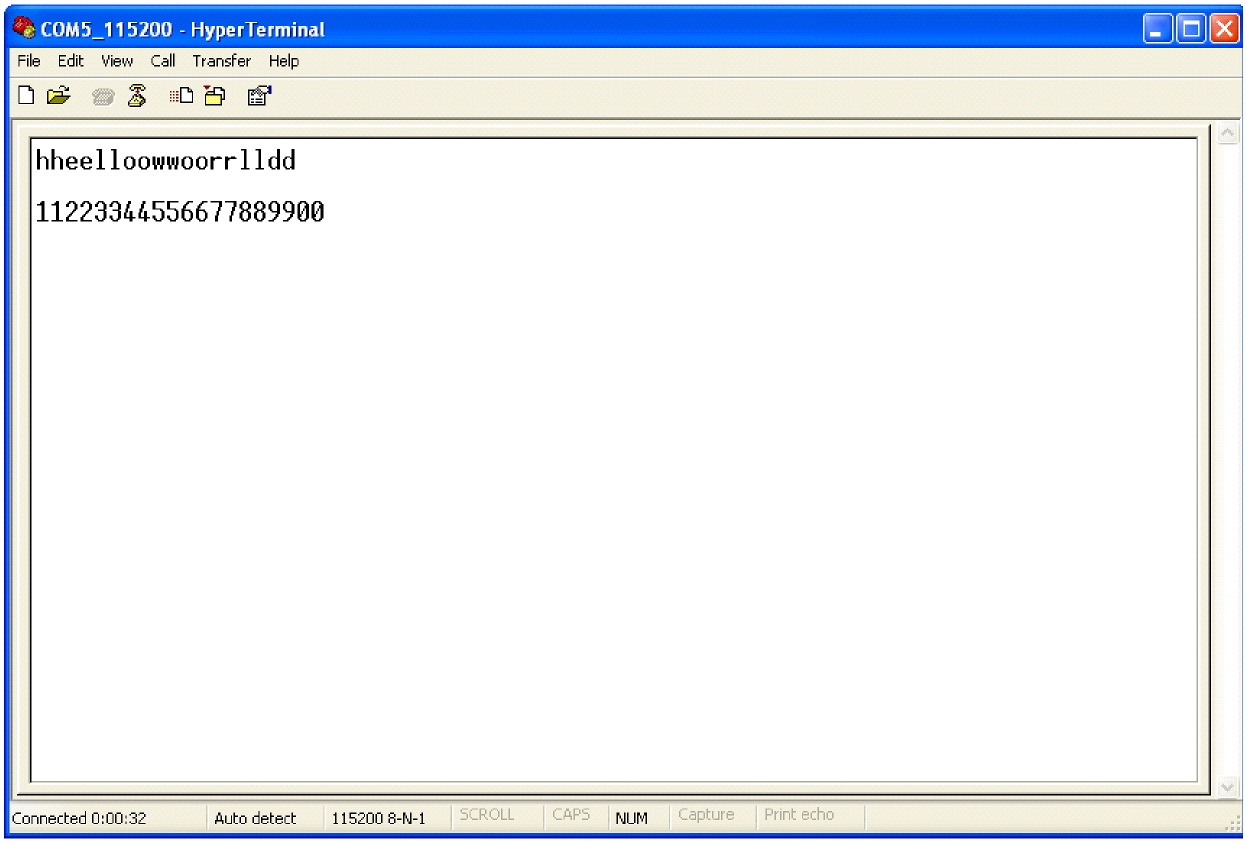
4. Plug-in the CDC device to the CDC Host (Board). The HyperTerminal displays a message as shown in Figure D-3.


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
Initialization passed. Plug-in CDC device to USB port.
Use ttyb: as the in/out port for CDC device data.
----- CDC control interface attach Event -----
State = attached  Class = 2  SubClass = 2  Protocol = 0
----- CDC data interface attach event -----
State = attached  Class = 10  SubClass = 0  Protocol = 0
----- CDC control interface selected -----
----- Device installed -----
----- CDC data interface selected -----
```

Figure D-3. Information of the device

5. Disconnect the COM1 from the PC. Connect the COM2 to the PC and try typing something. The result is echoed and displayed in the HyperTerminal.

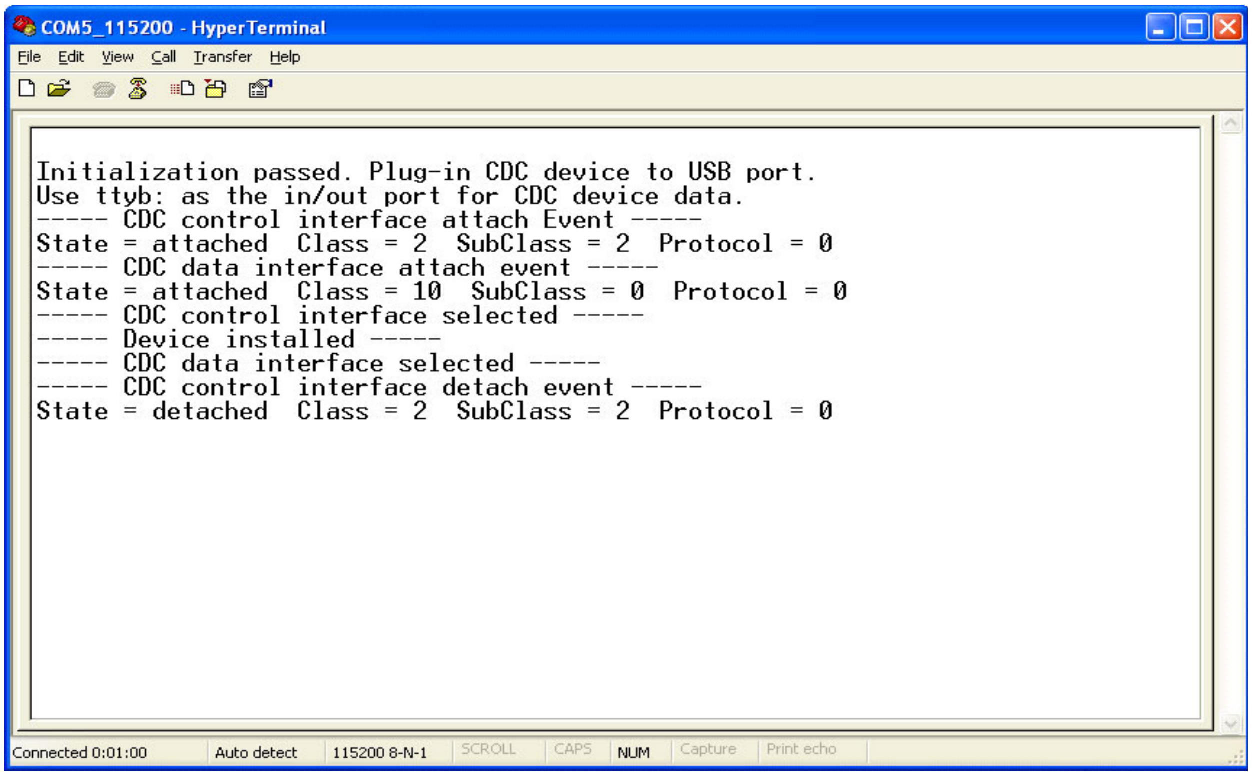

COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
hheelloowwoorrlldd
1122334455667788990
```

```
Connected 0:00:32    Auto detect    115200 8-N-1    SCROLL    CAPS    NUM    Capture    Print echo
```

Figure D-4. Character was echoed and display in HyperTerminal

6. Unplug the CDC device. The HyperTerminal is as shown in Figure D-5.


COM5_115200 - HyperTerminal
File Edit View Call Transfer Help

```
Initialization passed. Plug-in CDC device to USB port.
Use ttyb: as the in/out port for CDC device data.
----- CDC control interface attach Event -----
State = attached  Class = 2  SubClass = 2  Protocol = 0
----- CDC data interface attach event -----
State = attached  Class = 10  SubClass = 0  Protocol = 0
----- CDC control interface selected -----
----- Device installed -----
----- CDC data interface selected -----
----- CDC control interface detach event -----
State = detached  Class = 2  SubClass = 2  Protocol = 0
```

Connected 0:01:00 Auto detect 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

Figure D-5. CDC device detached

Appendix E Audio Host Demo

This chapter is a quick guide about using the USB Audio Host Demo software package. The demo application is used to control and communicate with Audio Devices. The operation of the demo depends on the Audio Device type:

- Speaker type (Audio Device with stream OUT supported): Sends audio data stream to the device.
- Microphone type (Audio Device with stream IN supported): Receives audio stream data from devices and plays it.

In both cases, the application supports sending specific requests to the Audio Devices such as Mute Control. To take you through this guide, the demos are illustrated by using a MCF52259 Demo board.

NOTE

The Audio Host Demo supports either audio data transmit interface or audio data receive interface over isochronous pipe. If the Audio Devices support multi-data interfaces, only the final audio data interface is supported.

E.1 Setting Up the Demo

Set up the connections as shown in Figure E-1.

8. Make the first USB connection between the PC, where the software is installed, and the 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.
9. Make the second connection between the Demo board and the PC to display the log of the Demo board.
10. Make the third connection between the Audio Device and the Demo board.
11. Make the fourth connection between a speaker and the Audio Device.
12. If the Audio Device is a microphone, the speaker is connected to Audio Host instead of Audio Device.

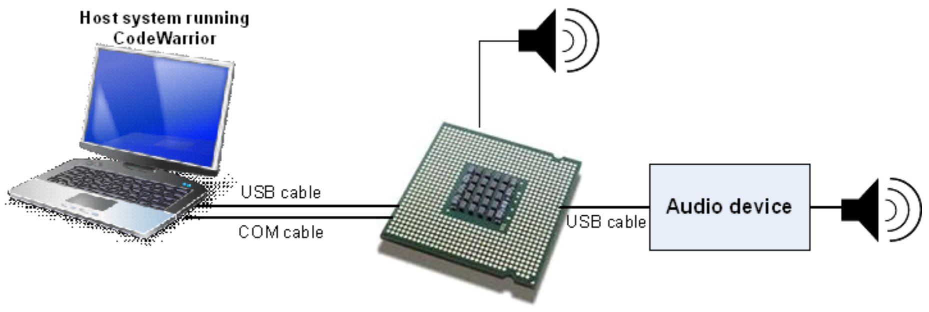
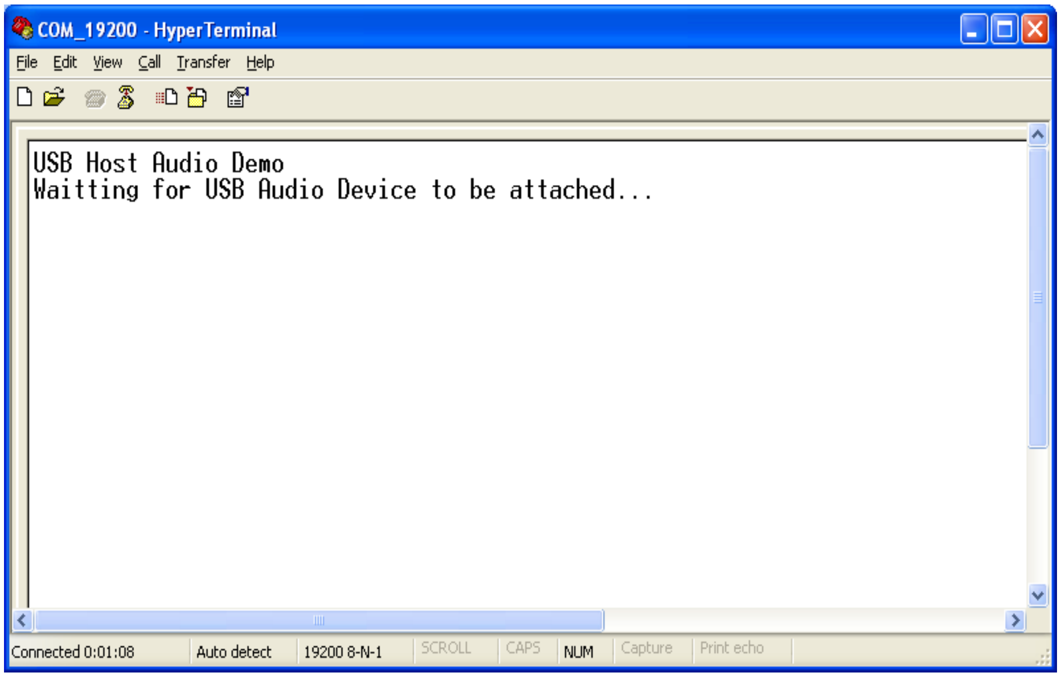


Figure E-1. Audio demo setup

E.2 Running the Demo

Perform the following steps to run the Audio Host Demo:

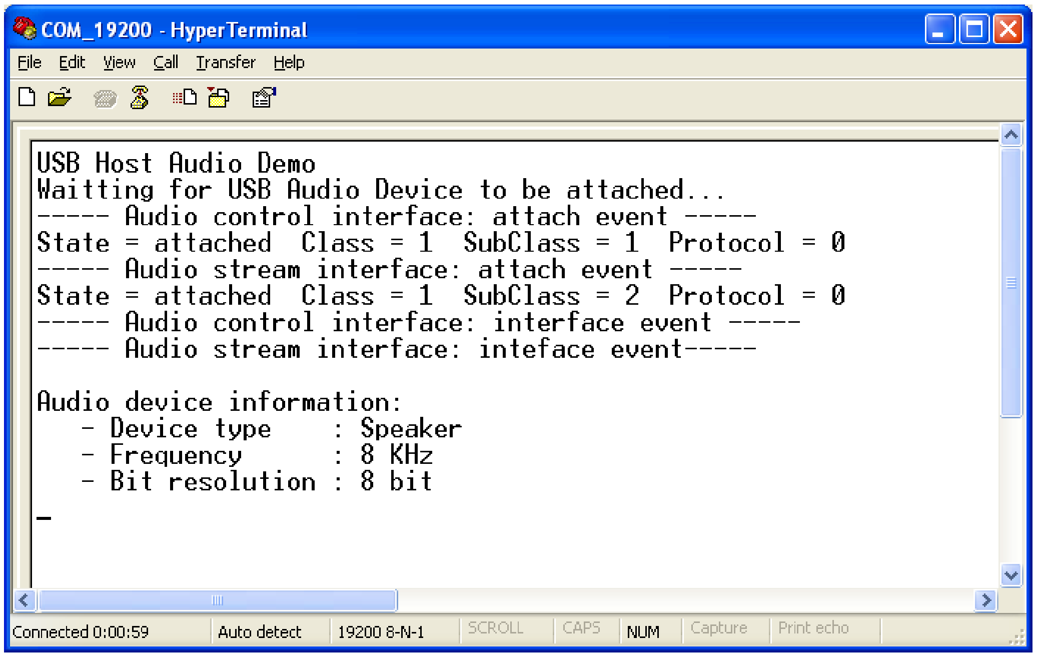
1. Open and load the image of the Audio Demo application to the board.
2. After the image has been loaded successfully, the HyperTerminal appears as shown in Figure E-2.



```
USB Host Audio Demo
Waitting for USB Audio Device to be attached...
```

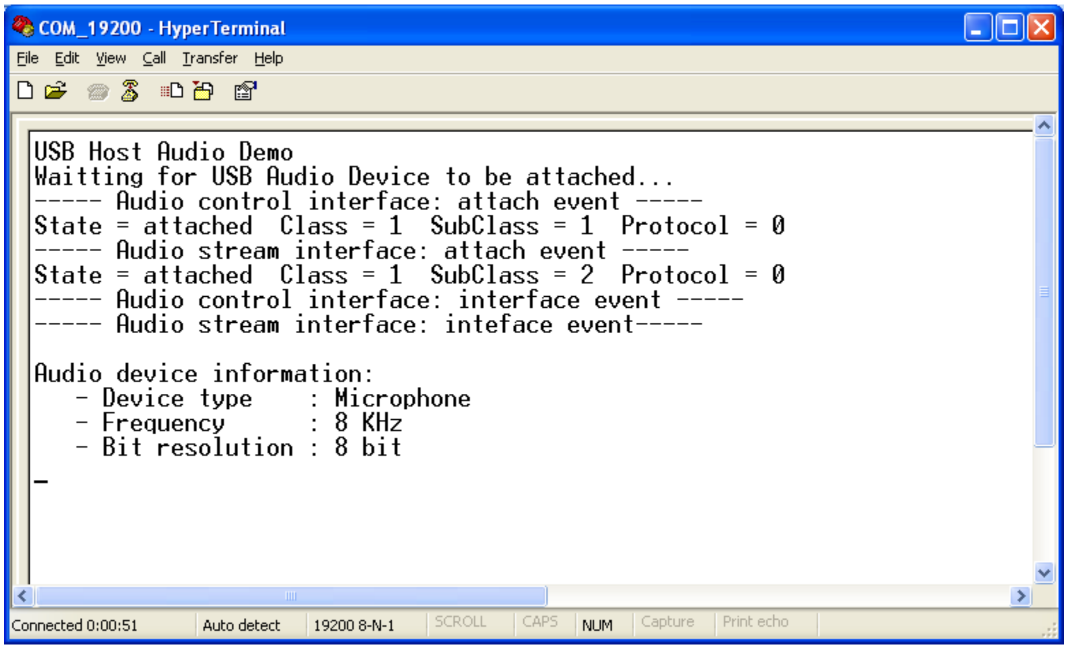
Figure E-2. USB Host waiting for audio device attachment event

3. Plug the Audio Device into the board. The Audio Device will be attached. If the Audio Device is of a speaker type, device information displays as shown in Figure E-3. If the Audio Device is of a microphone type, see Figure E-4.



```
USB Host Audio Demo
Waitting for USB Audio Device to be attached...
----- Audio control interface: attach event -----
State = attached  Class = 1  SubClass = 1  Protocol = 0
----- Audio stream interface: attach event -----
State = attached  Class = 1  SubClass = 2  Protocol = 0
----- Audio control interface: interface event -----
----- Audio stream interface: inteface event-----

Audio device information:
   - Device type    : Speaker
   - Frequency      : 8 KHz
   - Bit resolution : 8 bit
_
```

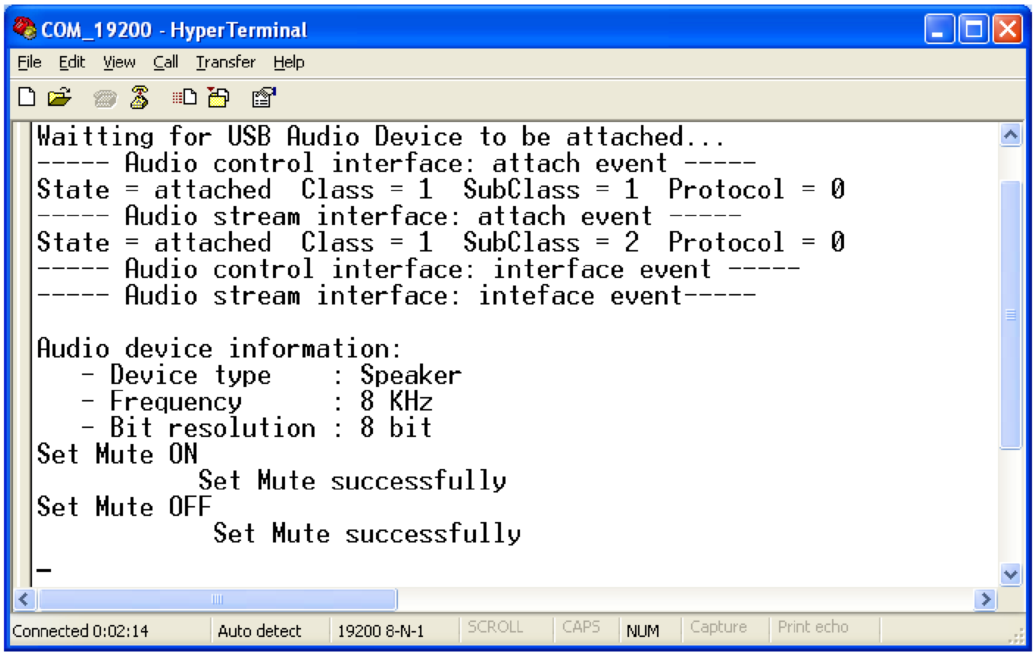
Figure E-3. Attached device is Speaker type



```
USB Host Audio Demo
Waitting for USB Audio Device to be attached...
----- Audio control interface: attach event -----
State = attached  Class = 1  SubClass = 1  Protocol = 0
----- Audio stream interface: attach event -----
State = attached  Class = 1  SubClass = 2  Protocol = 0
----- Audio control interface: interface event -----
----- Audio stream interface: inteface event-----

Audio device information:
   - Device type    : Microphone
   - Frequency      : 8 KHz
   - Bit resolution : 8 bit
_
```

Figure E-4. Attached device is Microphone type

4. Press Switch 1 to set Mute ON/OFF. The HyperTerminal screen appears as shown in Figure E-5.



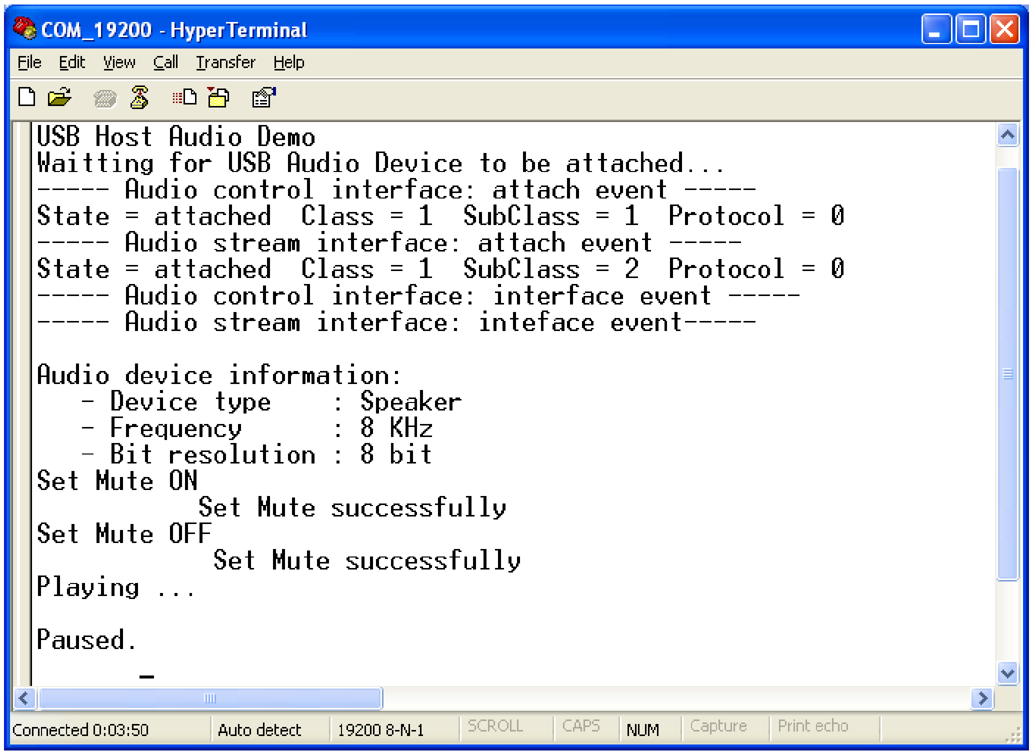
```
Waitting for USB Audio Device to be attached...
----- Audio control interface: attach event -----
State = attached  Class = 1  SubClass = 1  Protocol = 0
----- Audio stream interface: attach event -----
State = attached  Class = 1  SubClass = 2  Protocol = 0
----- Audio control interface: interface event -----
----- Audio stream interface: inteface event-----

Audio device information:
   - Device type     : Speaker
   - Frequency       : 8 KHz
   - Bit resolution : 8 bit
Set Mute ON
           Set Mute successfully
Set Mute OFF
            Set Mute successfully
_
```

Figure E-5. Set Mute ON/OFF

5. Press Switch 2 to Start/Stop transferring audio data stream between the Audio Host and the Audio Device.

- If the attached device is of a speaker type, you will be able to hear the sound from the speaker which is connected to the Audio Device.
- If the attached device is of a Microphone type, you will be able to hear the sound from the speaker which is connected to the Audio Host.

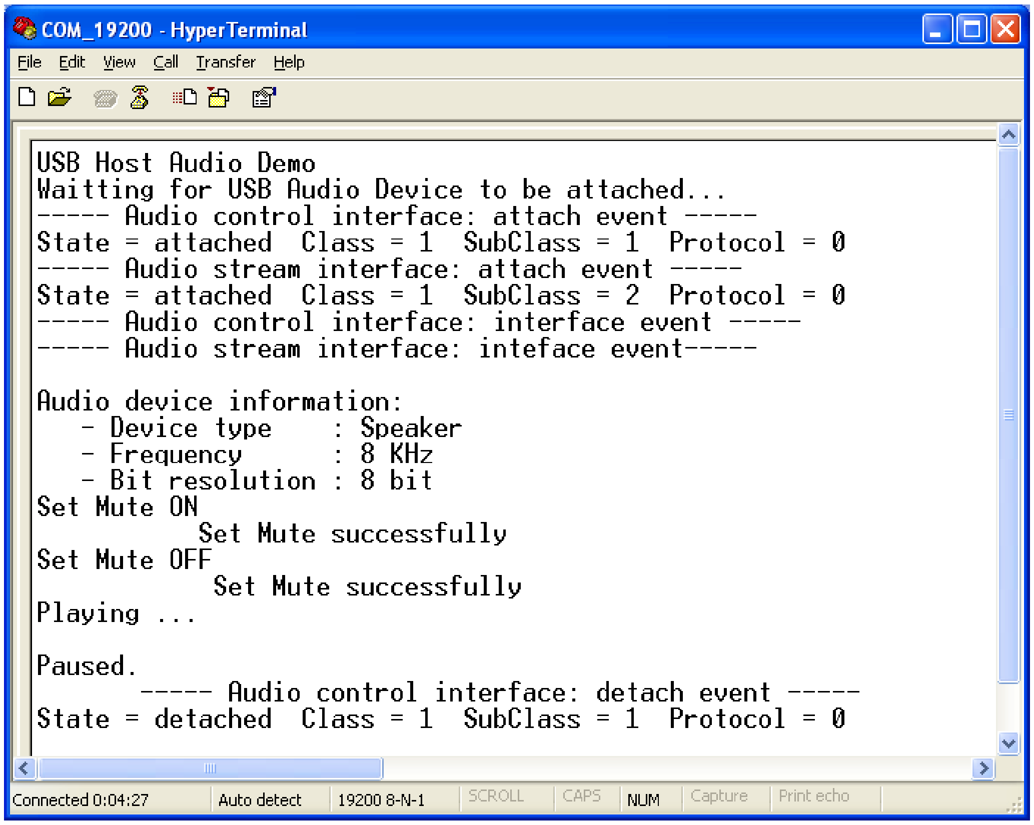
The HyperTerminal screen appears as shown in Figure E-6.



```
USB Host Audio Demo
Waitting for USB Audio Device to be attached...
----- Audio control interface: attach event -----
State = attached  Class = 1  SubClass = 1  Protocol = 0
----- Audio stream interface: attach event -----
State = attached  Class = 1  SubClass = 2  Protocol = 0
----- Audio control interface: interface event -----
----- Audio stream interface: inteface event-----

Audio device information:
   - Device type     : Speaker
   - Frequency       : 8 KHz
   - Bit resolution : 8 bit
Set Mute ON
           Set Mute successfully
Set Mute OFF
            Set Mute successfully
Playing ...

Paused.
       _
```

Figure E-6. Start/Stop transferring audio data

6. Unplug the Audio Device. The HyperTerminal displays a message as shown in Figure E-7.



```
USB Host Audio Demo
Waitting for USB Audio Device to be attached...
----- Audio control interface: attach event -----
State = attached  Class = 1  SubClass = 1  Protocol = 0
----- Audio stream interface: attach event -----
State = attached  Class = 1  SubClass = 2  Protocol = 0
----- Audio control interface: interface event -----
----- Audio stream interface: inteface event-----

Audio device information:
   - Device type     : Speaker
   - Frequency       : 8 KHz
   - Bit resolution  : 8 bit
Set Mute ON
           Set Mute successfully
Set Mute OFF
            Set Mute successfully
Playing ...

Paused.
       ----- Audio control interface: detach event -----
State = detached  Class = 1  SubClass = 1  Protocol = 0
```

Figure E-7. Audio Device detached