Freescale MQX™ USB Host User’s Guide

Document Number: MQXUSBHOSTUG
Rev. 2
4/2011

LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: MQXUSBHOSTUG
Rev. 2
4/2011

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 http://www.freescale.com/mqx.

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 release added into the document.

Chapter 1 Before you begin

1.1 About This Book

This book describes the Freescale MQX™ USB Stack Architecture. 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:

- Chapter 1 — Before you begin
- Chapter 2 — Get Familiar Firstt
- Chapter 3 — Design Overview
- Chapter 4 — Developing Applications
- Chapter 5 — Debugging Applications
- Chapter 6 — Key Design Issues

1.2 Reference Material

As recommendation consult the following reference material:

- *Universal Serial Bus Specification Revision 1.1*
- *Universal Serial Bus Specification Revision 2.0*
- *OTG supplement to USB 2.0 specifications Revision 1.3*

Chapter 2 Get Familiar First

2.1 Introduction

Freescale Semiconductor provides Freescale MQX™ USB Host/Device Stack operating at both low-speed and full speed. Freescale MQX™ RTOS comes with support of a complete software stack combined with basic core drivers, class drivers, and plenty of sample programs that can be used to achieve the desired target product. This document intends to help customers develop an understanding of USB stack in addition to useful information on developing and debugging USB applications. The document also addresses some daily development issues such as software timing response constraints under USB and debugging help provided by RTOS. No amount of documentation can replace the need to play with the source code. 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 that are familiar with basic terms on USB and are involved with direct development of the product on USB Host/Device Stack. The following list[1] of documents are useful for developers when developing a deeper understanding of USB Host/Device stack and while debugging the USB applications during product development.

- USB Host API
- USB Device API
- USB OTG API

2.2 Software Suite

Freescale MQX™ RTOS customers may have one of the following three options of the USB Host/Device Stack.

- Device only full speed software
- Host only full speed software
- OTG full speed software

NOTE

OTG full speed software/examples is not part of the current MQX release and will be added in future MQX versions.

These options are compiled from a complete USB Host/Device stack tree that can be seen Figure 2-1 Separation between host and device software directories are by names. Additional subdirectories exist in the source structure and contain specific code to particular hardware implementation. An example may be the khci for the MCF52259 host controller interface.

1. Refer to Freescale MQX™ release notes for current supported features.

This document is written for complete software architecture. A reader can skip a part of the design that is not applicable for his particular software.

2.2.1 Directory Structure

The following bullets display the complete tree of Freescale MQX™ USB Host/Device Stack. For example, if a customer makes a device only product, there is concern only about the following directories.

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

Similarly a customer who builds a host only product is concerned with the following directories.

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

If a customer is building an OTG application, there is concern about both host and device functions.

Table 2-1 briefly explains the purpose of each of the directories in the source tree. Those who are new to the USB Host\Device stack can use this table for reference.

Table 2-1. Directory Structure

Directory Name	Purpose
Usb	Root directory of the software stack
usb\build	`*.mcp` - for building the complete project.
Freescale MQX\lib	Output directory of the build
usb\host\examples	Host-side class-driver examples
usb\device\examples	Device firmware development examples
usb\otg\examples	OTG core usage 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\rtos usb	RTOS thin layer sources
Usb\host\rsource\rtos\mqx	RTOS layer source

Figure 2-1 shows the default directory structure.

NOTE

Figure 2-1 may not represent the latest tree but can give an idea of the USB Host\Device stack structure.

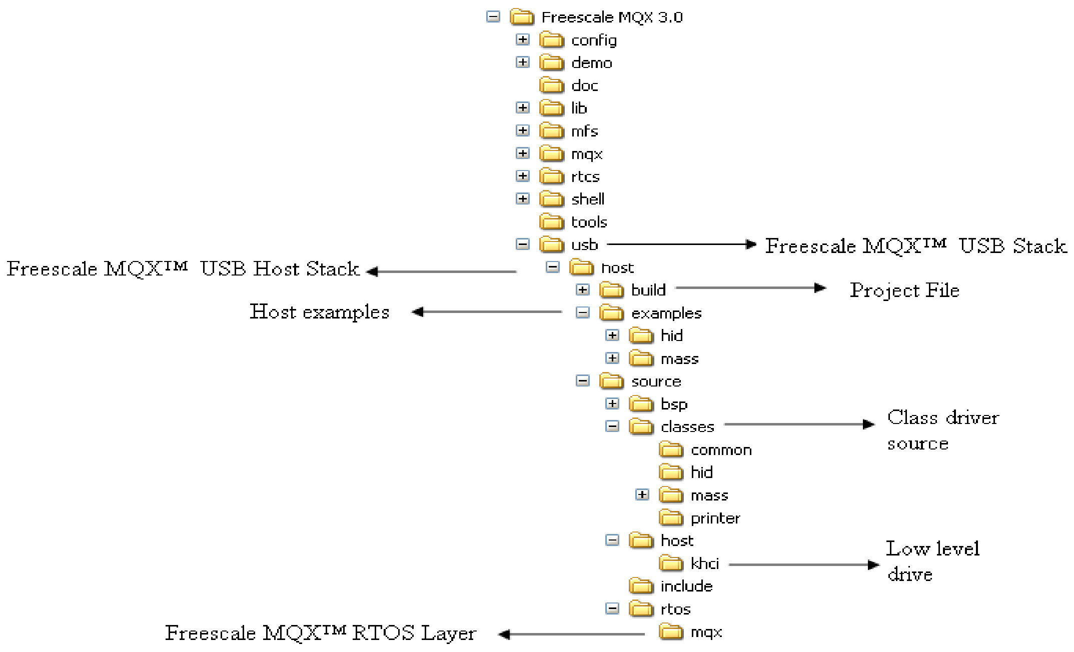


Figure 2-1. Default Directory Structure

Chapter 3 Design Overview

3.1 Design Goals

Freescale MQX™ USB Host\Device 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 minimum code size and minimum components required for the product design. USB Host\Device stack is designed to be flexible in architecture in a way that undesired components can be taken away from the software suite. It remains a constant goal from the USB Host\Device stack team to add the extra functionality as a plug in module that can be completely eliminated at compile time.

3.1.2 Hardware Abstraction

One goal of the USB Host\Device stack is to provide true hardware independence in the application code. The USB Host\Device stack is provided with 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\Device 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\Device stack is designed to function under a single thread with minimum interrupt latency. If looking at the code underneath, notice the USB Host\Device 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 of a host and one or more devices that respond to the host request. Because of the introduction of OTG specification, it is possible to achieve both host and device roles on the same hardware via the software manipulation. Figure 3-1 shows how a hypothetical customer OTG or device application may be designed.

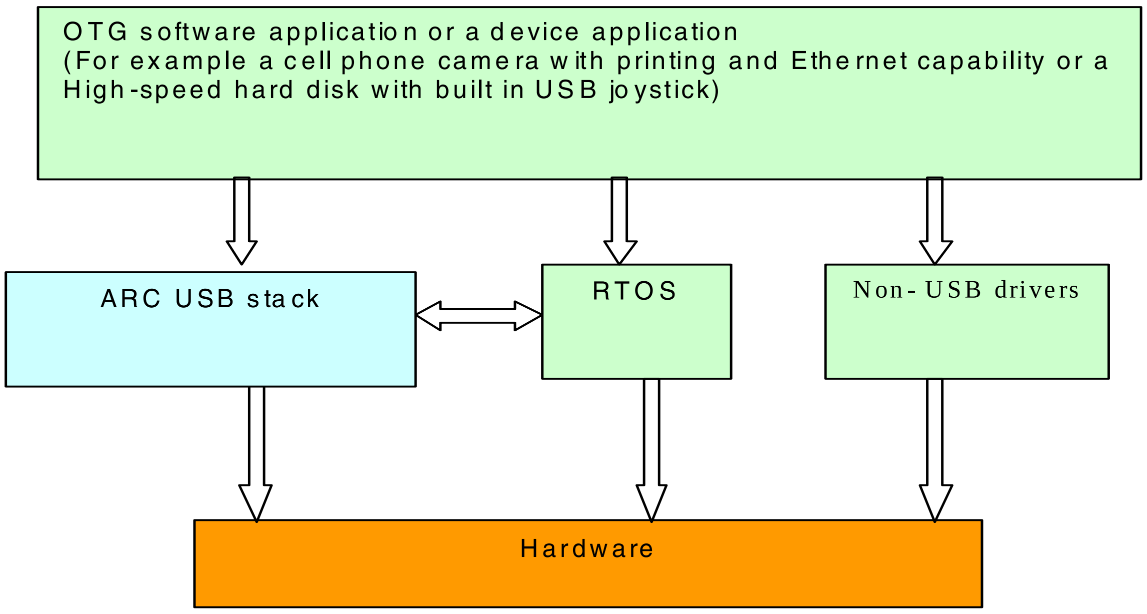

OTG software application or a device application
(For example a cell phone camera with printing and Ethernet capability or a High-speed hard disk with built in USB joystick)
ARC USB stack
RTOS
Non-USB drivers
Hardware

Figure 3-1. Hypothetical Target Design

The customer application can communicate with USB hardware using Freescale MQX™ USB Host\Device stack. Figure 3-2 goes deeper inside the stack design.

3.2.1 Complete USB Stack Diagram

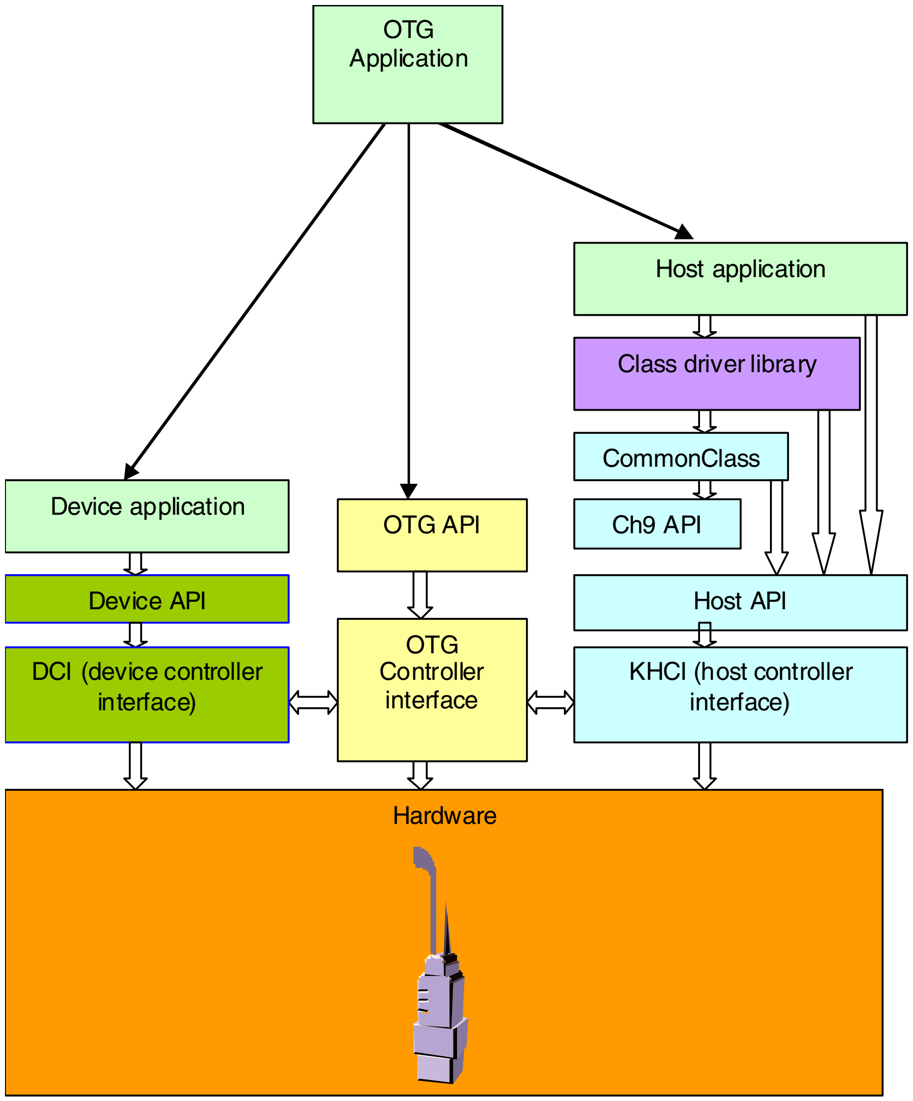


Figure 3-2. Complete USB Stack Diagram

Notice three applications in Figure 3-2. Those who are developing OTG applications have to develop both device and host applications, and write a central OTG application that works with the OTG API functions to decide the role (host or device) and load the appropriate device or host applications. Those who are making device-only or host-only applications do not have to be concerned about OTG API. The next few sections explain each of the parts of the stack in detail.

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 using logical pipes. After the device is enumerated, a software application needs the capability 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 host and device is done using logical pipes that are represented in the stack as pipe handles. Figure 3-3 shows the description of each of the blocks shown as part of the host API.

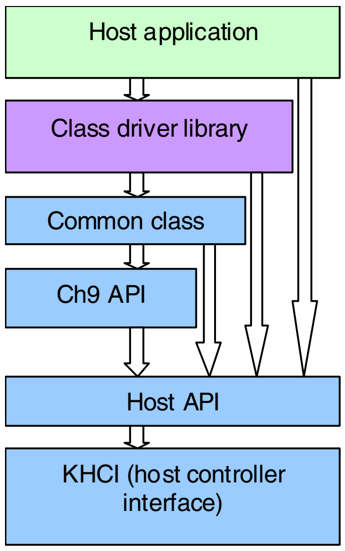


Figure 3-3. Freescale MQX™ USB Host Stack

3.3.1.1 Host Application

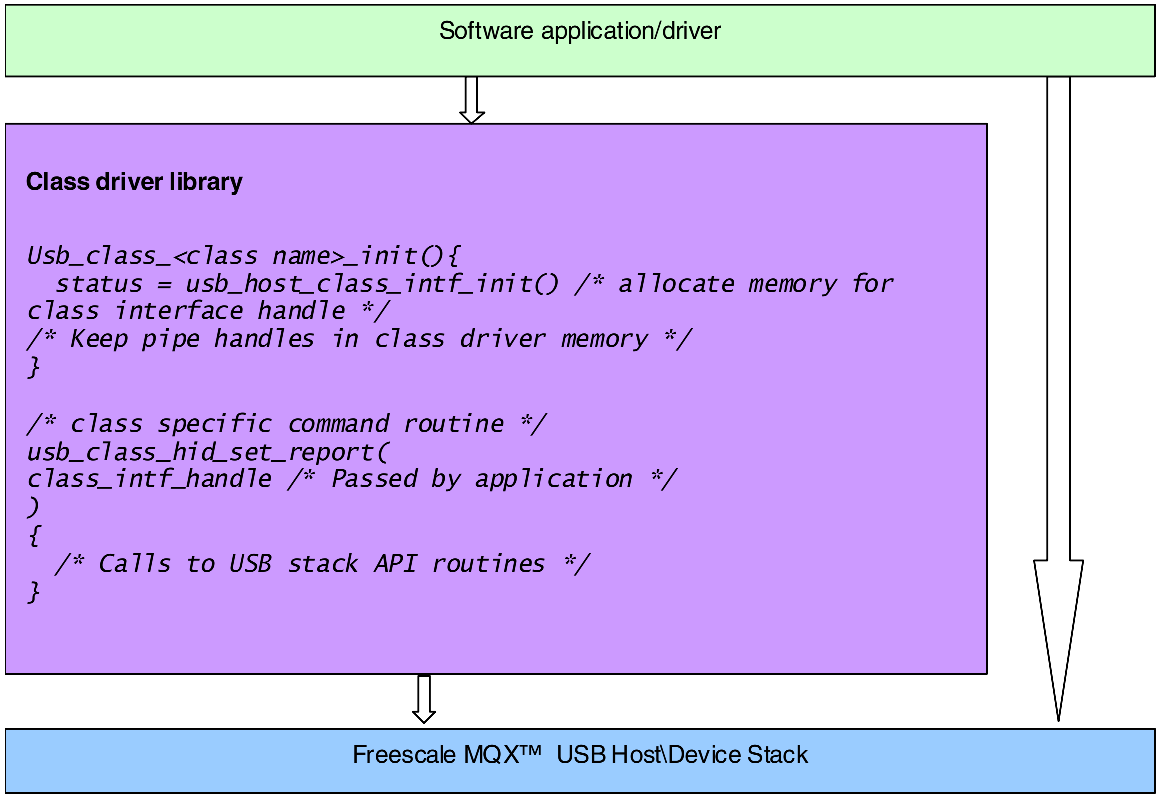
A host embedded application software (also called a device driver) is implemented for a target device or a class of device. Freescale MQX™ USB Device Stack comes with examples for a few classes of devices. Application software can use API routines to start and stop communication with a USB bus. It can suspend the bus or wait for a resume signal for the bus. Please 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 into the application. These routines implement standard functionality of the class of device, defined by 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 talk to all the devices. Storage devices mainly follow universal floppy interface (UFI) specifications making it easy to write an application that can talk to most storage devices.

The USB Host\Device stack comes with a class library for UFI commands that can be called to talk to 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\Device stack and does not necessarily have to be used.

The design of the class driver libraries in the USB Host\Device stack follows a standard template. All libraries have an initialization routine called by the USB Host\Device 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.


Software application/driver
Class driver library
Freescale MQX™ USB Host\Device Stack

```
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 the USB mass-storage commands are comprised of three phases:

- CBW
- Data
- CSW

For details of these phases see reference document on 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\Device 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 the 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 down to the library using a routine that can pass the command down to the USB.

All the commands have a single-function entry point in the mass-storage API. However, all the 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 the 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 the **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 for better understanding of the code flow.

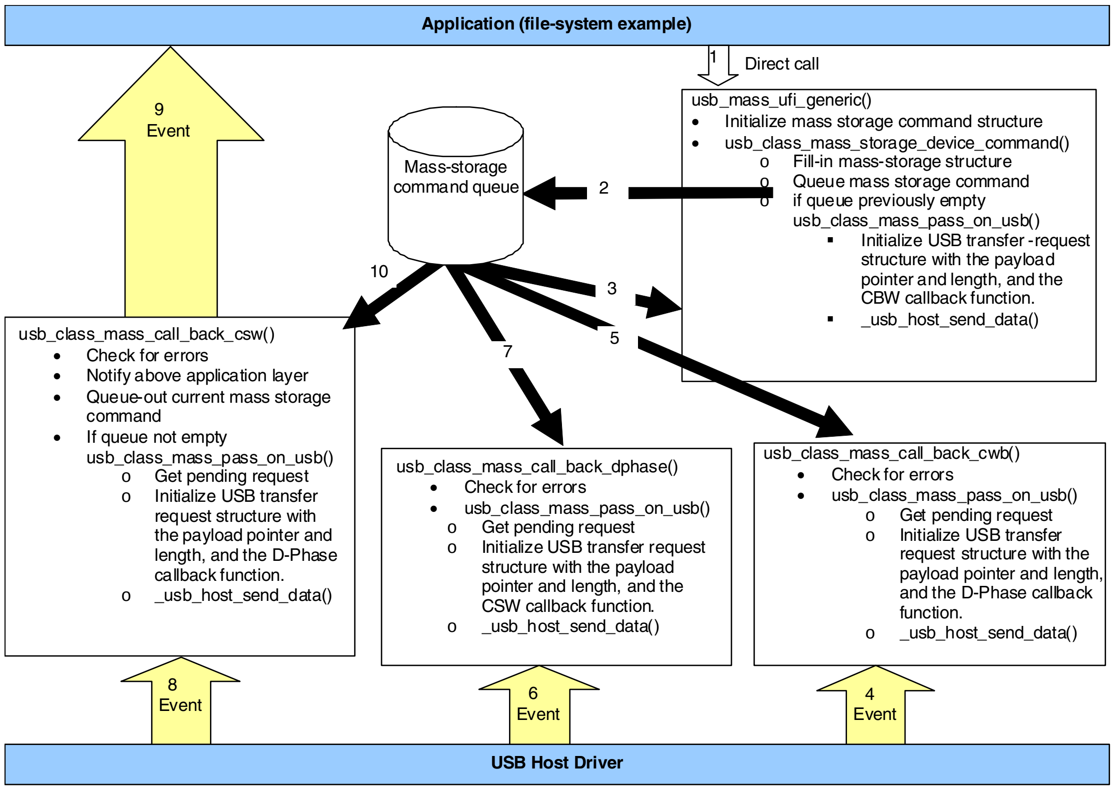

Application (file-system example)
1
Direct call
usb_mass_ufi_generic()
• Initialize mass storage command structure
• usb_class_mass_storage_device_command()
o Fill-in mass-storage structure
o Queue mass storage command
o if queue previously empty usb_class_mass_pass_on_usb()
▪ Initialize USB transfer-request 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()
o Get pending request
o Initialize USB transfer request structure with the payload pointer and length, and the D-Phase callback function.
o _usb_host_send_data()
usb_class_mass_call_back_dphase()
• Check for errors
• usb_class_mass_pass_on_usb()
o Get pending request
o Initialize USB transfer request structure with the payload pointer and length, and the CSW callback function.
o _usb_host_send_data()
usb_class_mass_call_back_cwb()
• Check for errors
• usb_class_mass_pass_on_usb()
o Get pending request
o Initialize USB transfer request structure with the payload pointer and length, and the D-Phase callback function.
o _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 and by looking at the API document it is difficult to say what routines belong to this layer. It is a deliberate design attempt to reuse routines to minimize the code size.

Routines inside the common-class layer take advantage of the fact that in USB all devices are enumerated with the same sequence of commands. When a 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 on how an application handles a plugged-in device. Figure 3-6 illustrates how device plugin works.

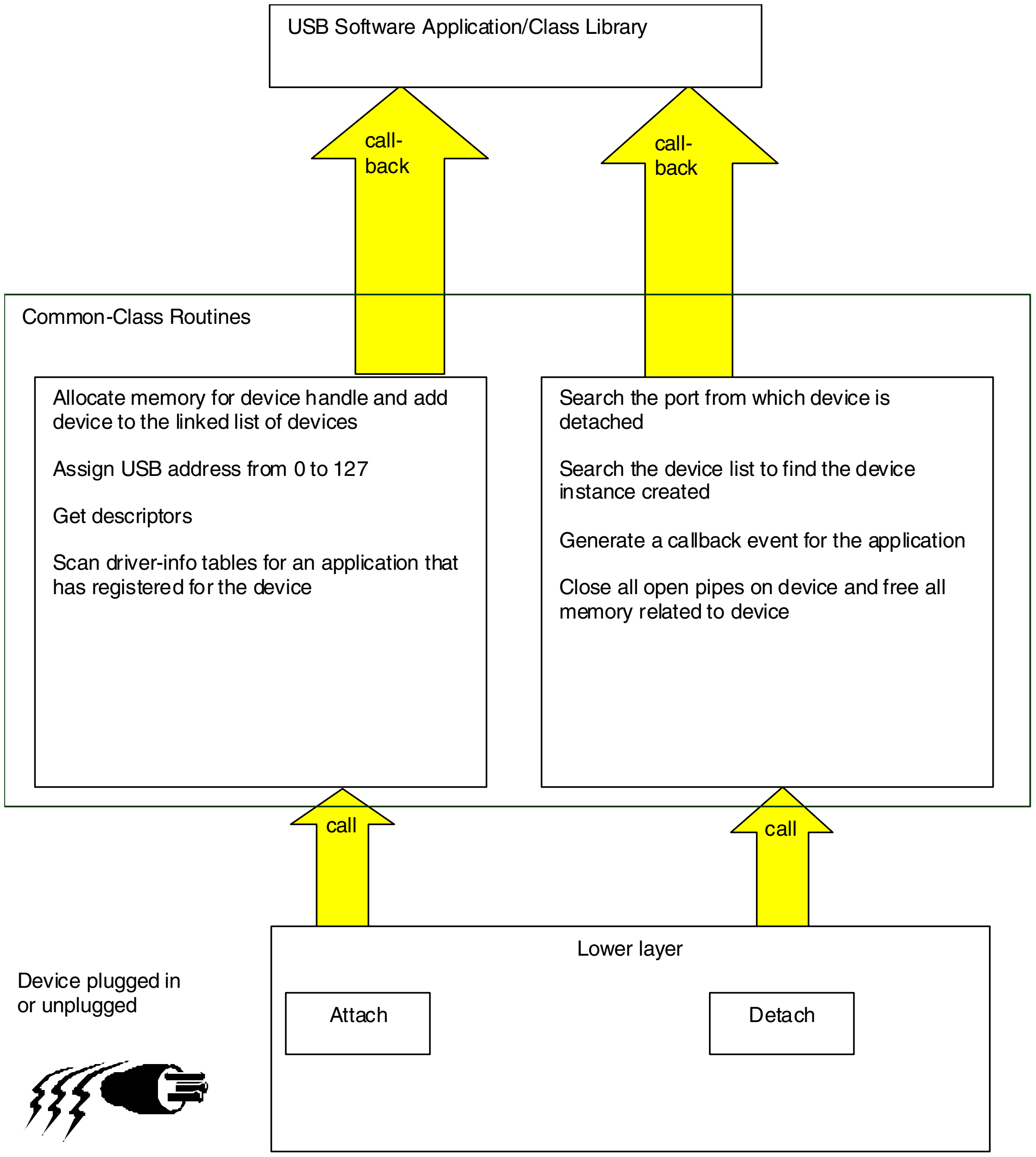

USB Software Application/Class Library
call-back
call-back
Common-Class Routines
Allocate memory for device handle and add device to the linked list of devices
Assign USB address from 0 to 127
Get descriptors
Scan driver-info tables for an application that has registered for the device
Search the port 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

The USB specification document has Chapter 9 that 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 Chapter 9 commands. All customer applications can be written to use only this API and without the common-class API or class libraries.

The USB Chapter 9 commands are outside the scope of this document and it requires a good familiarity with 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. Please see the 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\Device 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. This is used in freeing all memory when 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.

A good understanding of the source inside the API layer can be developed by reading the API routine and tracing it down to the hardware drivers.

3.3.1.8 KHCI (Host Controller Interface)

KHCI is a completely hardware-dependent set of routines responsible for queuing and processing of USB transfers and searching for hardware events. Source understanding of this layer requires understanding of hardware.

3.3.2 Device Overview

Freescale MQX™ USB Host\Device stack comes with support for low-speed and full-speed device cores. By nature of USB system, a device application is a responder to the host commands. This is a simpler application in comparison to host driver applications. The USB Host\Device stack is a two layer API that provides sufficient 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 down to the hardware. This is a summary of all the functions that are implemented by this layer.

- Registration and unregistration of services with the USB Host\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.

Conceptually, all the 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. Figure 3-7 shows how this works.

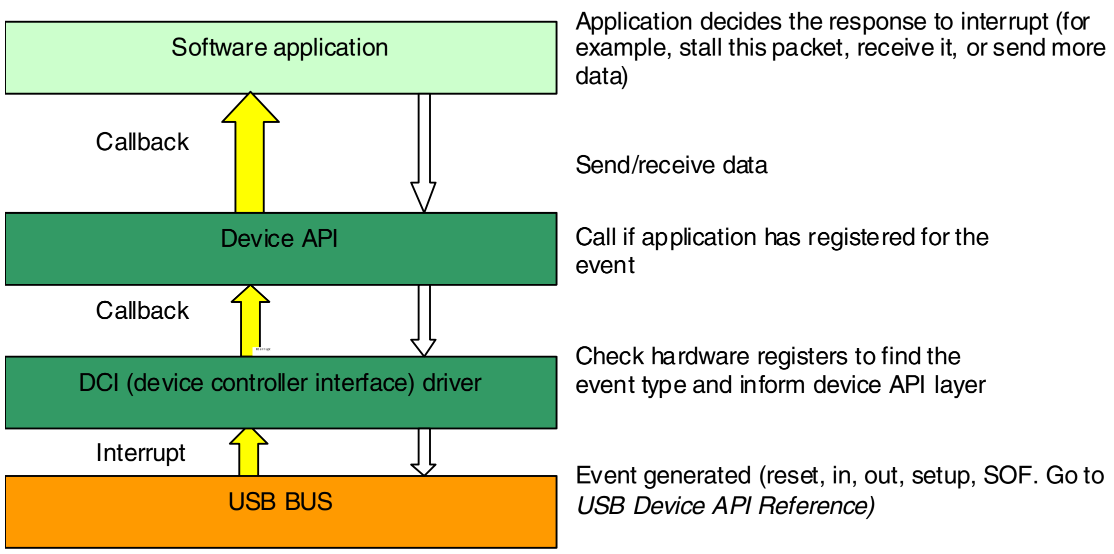
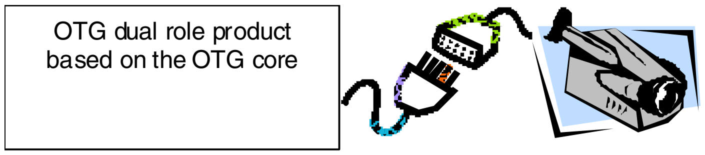

Software application
Callback
Device API
Callback
DCI (device controller interface) driver
Interrupt
USB BUS
Application decides the response to interrupt (for example, stall this packet, receive it, or send more data)
Send/receive data
Call if application has registered for the event
Check hardware registers to find the event type and inform device API layer
Event generated (reset, in, out, setup, SOF. Go to *USB Device API Reference)*

Figure 3-7. Device Interrupt and Response

3.3.3 OTG Overview

The OTG specification from USB requires the implementation of a complex state machine inside the software and hardware. Under OTG rules, OTG hardware switches states based on several factors such as watchdog timers, bus-voltage levels, cable-connection changes (called ID change), and certain protocol events such as SRP and HNP. This requires the software to provide a proper response to the state-change event and sometimes switching over the role from host to device or device to host. The software can also decide to initiate HNP and switch roles. The USB Host\Device stack implements the complete OTG state machine with a well-defined API for application software. Application software can use the USB Host\Device stack OTG API to find the current state of the OTG core and make decisions. Applications register for events such as Host UP or Device UP to find when to switch roles.

Software can also control certain variables to change the state of OTG controller. The following may clarify some OTG events. Any discussion on OTG can be fairly technical and requires a solid understanding of the OTG state machine. The USB committee clarified the complexity of the state machine.



OTG software may be in one of the following situations. The software application must be prepared to handle all situations.

Table 3-1. OTG Situations and Repective Software Action

Situation	Possible Software Action (see source Examples)
OTG event (No cable connected)	Starts in device mode as default.
OTG event (A side of cable connected, but no device)	Starts Host stack and waits for the attached interrupt.
OTG event (A side of cable connected with a non-OTG device)	Starts the Host stack, handles an attach interrupt from the device and starts normal USB communication with device.
OTG event (A side of cable connected with a dual-role device)	Starts as a Host stack like above, but looks if this device is supported or not. If the device is not supported, starts HNP by using OTG API routines.
OTG event (B side of cable connected, but no device)	Starts in device mode as default.
OTG event (B side of cable connected with a non-OTG device)	Switches over to *B_HOST* state and starts communicating with the device.
OTG event (B side of cable connected with a dual-role device)	Switches over to *B_HOST* state and starts communicating with the device. If the device is not supported, provides an opportunity to do HNP.
OTG event (SRP started from the other side of the cable)	Handles SRP active event and resumes Host operation.
OTG event (HNP started from the other side of the cable)	Switches over roles from existing mode to another mode.
OTG event (cable disconnected)	Handles detach interrupt and mode to default device mode.

Freescale MQX™ USB Host\Device stack software generates the following main events for software application. Software application can decide, which of the above events are taking place by looking for the following events in combination of querying the state of OTG state machine by using the API routines.

Table 3-2. Main Events Generated by Host/Device Stack

Event Generated	Why generated?
USB_OTG_DEVICE_DOWN	Application software started HNP by calling the **usb_otg_set_status()** routine or the other side of the cable started taking over host role under HNP.
USB_OTG_DEVICE_UP	Default when software is started.
USB_OTG_HOST_DOWN	Cable disconnected or under HNP.
USB_OTG_HOST_UP	A cable connected or under HNP.
USB_OTG_SRP_ACTIVE	The other side of the cable started SRP.
USB_OTG_SRP_FAILED	Software application started SRP, but the attempt failed with the other side of the cable.
USB_OTG_HNP_FAILED	Software application started HNP, but the attempt failed with the other side of the cable.

To supplement the application decision, the following API routines have been provided.

Table 3-3. Main Events Generated by Host/Device

Routine Name	Purpose
Usb_otg_get_status()	It is used by software application to find information from OTG stack such as the current state of OTG state machine. For example, in the device mode, if cable is disconnected, software application can query the stack to find if it is in *B_IDLE* state. If it is, it may decide to keep certain operations off.
_Usb_otg_set_status()	OTG specifications provide certain parameters that can be controlled by software such as *A_SUSPEND_REQ* is set to true, when software applications in host mode wants to suspend the bus and provide an opportunity to other device to take over the role of host. These parameters correspond to certain hardware controls (see OTG specifications for details). This routine can be used to control such parameters. Freescale MQX™ Host\Device stack comes with plenty of completely tested examples that demonstrate how to make use of this routine. Firmware programmers can reuse most of the code in their applications.

Only a few lines of code is required to make an up and running OTG application software. See some of the source code examples for development of better understanding of OTG events and API routine calls.

Chapter 4 Developing Applications

4.1 Compiling Freescale MQX™ USB Host/Device Stack

4.1.1 Why Rebuild USB Host/Device Stack

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

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

CAUTION

It is not recommend modifing USB Host/Device stack data structures. If so, some of the components in the Precise Solution™ Host Tools family of host software-development tools may not perform correctly. Modify USB Host/Device stack data structures only if experienced with the USB Host/Device stack.

4.1.1.1 Before Beginning

Before rebuilding the USB host/device stack:

- Read the MQX User Guide document for MQX RTOS rebuild instructions. A similar concept also applies to the USB Host/Device stack.
- Read the MQX release notes that accompany Freescale MQX 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/device stack directory structure, re-build instructions as described in the release notes document and the instructions provided in the following sections

4.1.1.2 USB Directory Structure

The following table shows the USB Host/Device 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/Device stack within the Freescale MQX distribution
\build	
\codewarror	CodeWarrior-specific build files (project files)
\examples	
\example	Source files (.c) for the example and the example's build project.
\source	All USB Host/Device stack source code files
\lib	
\<board>.<comp>\usb	USB Host/Device stack library files built for hardware and environment

4.1.1.3 USB Host/Device Stack Build Projects in Freescale MQX

The USB Host/Device stack build project is constructed like 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/Device stack code is not specific to any particular board nor to a processor derivative, a separate USB Host/Device 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 this board-independent code is built into the board-specific output directory is because 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 User Guide for more details

4.1.1.3.1 Post-Build Processing

All USB Host/Device 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/Device 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/Device stack services do not need to make any reference to the USB Host/Device stack source tree.

4.1.1.3.2 Build Targets

The CodeWarrior development environment enables multiple build configurations named build targets. All projects in the Freescale MQX USB Host/Device 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 exampple, lib\m52259evb.cw\usb\ usb_hdk.a).

4.1.1.4 Rebuilding Freescale MQX USB Host/Device Stack

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

For specific information about rebuilding MQX USB Host/Device stack and the example applications, see Freescale MQX release notes .

4.2 Developing OTG Applications

An OTG application is a dual role application that acts like a host for devices and like a device for hosts. A practical example of an OTG application can be a file system example (provided with the USB Host\Device stack) that acts like a storage disk for a PC and allows the storage devices to connect and run a file system over it. Another example can be a digital camera application that acts like a disk for the host and allows printing files to a printer.

4.2.0.1 Define a File List Convention

When developing an OTG application, it is good to have a clear distinction between device and host code. The OTG dual role applications that come with the USB Host\Device stack, in general define at least three files, appname_d.c, appname_h.c, appname.c. For example `demo_d.c` defines all the code that makes the application act like a device, `demo_h.c` for the host role code, and `demo.c` as the main file that starts the application with common and necessary steps such as of the OTG controller initialization.

4.2.0.2 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/device stack generates an attached callback when a device matches this table entry. The application now can communicate to the device. The following structure defines one member of the table. If a Vendor – Product pair does not match for a device, Class – SubClass, and Protocol is checked to match. Use 0xFF in SubClass and Protocol struct member to match any SubClass / Protocol.

```
/* structure to define information for one class or device driver */
typedef struct driver_info
{
   uint_8         idVendor[2];      /* Vendor ID per USB-IF */
   uint_8         idProduct[2];     /* Product ID per manufacturer */
   uint_8         bDeviceClass;     /* Class code, 0xFF if any */
   uint_8         bDeviceSubClass;  /* Sub-Class code, 0xFF if any */
   uint_8         bDeviceProtocol;  /* Protocol, 0xFF if any */
   uint_8         reserved;         /* Alignment padding */
   event_callback attach_call;
} USB_HOST_DRIVER_INFO, _PTR_ USB_HOST_DRIVER_INFO_PTR;
```

The following is a sample driver info table. See the example source code for samples. Notice 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.

```
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_MOUSE,       /* Protocol                         */
      0,                            /* Reserved                         */
      usb_host_hid_mouse_event      /* Application call back function   */
   },
   {
      {0x00, 0x00},                 /* All-zero entry terminates        */
      {0x00, 0x00},                 /* driver info list.                */
      0,
      0,
      0,
      0,
      NULL
   }
};
```

NOTE

This is not the topic of discussion in this document, but it is a recommended practice. Initialization can be done before proceeding initialization of the USB host\device stack. After USB host\device stack is initialized, interrupts start getting generated and time constraints of the USB start operating. This functionaly can impact performance in other initializations.

4.2.1 Initializing the OTG Controller

This operation in not performed by the host. The application is responsable of it. The first step in the firmware code is to communicate to the USB host\device stack that the application is a dual role. This allows to register a callback routine to listen for OTG events. See the OTG API document for a complete list of events. These OTG events allow the software application to determine its role as host or device at run time and to take proper actions. Under OTG specification, it is the hardware that decides the role and the USB cable connectors decide the role of the software. All software applications can be ready to switch roles based on the events generated by the hardware. For an example, if an OTG event called HOST_UP then means that the host role must be activate. As a result, the device role is deactivated. All the memory and structures used as a device are free. The host initialization routine is called to start acting as Host. The initialization of OTG is done by filling a data structure and calling a init routine to initialize the OTG stack. The following code demonstrates the OTG controller intialization in a default device mode.

```
/* default configuration: A = host, B = peripheral */
   otg_init.A_BUS_DROP  = FALSE;
   otg_init.A_BUS_REQ   = TRUE;
   otg_init.B_BUS_REQ   = FALSE;
   otg_init.SERVICE     = otg_service;

   /*
   ** Initialize the USB interface, causing a call back to the otg_service
   ** routine
   */
   status = _usb_otg_init(0, &otg_init, &otg_handle1);
```

4.2.2 Receiving OTG Events

Once the OTG stack has been initialized, the stack generates callbacks or events that need to be handled by the software application. An issue of concern is that these callbacks are generated at an interrupt level and should be focused for minimum amount of processing and limited or no debugging code. The following example code shows what is done under a USB_OTG_HOST_UP event callback.

```
Void otg_service {
switch (event) {
      case USB_OTG_HOST_UP:

         /*
         ** It means that we are going to act like host, so we initialize the
         ** host stack. This call will allow USB system to allocate memory for
         ** data structures, it uses later (e.g pipes etc.).
         */
         status = _usb_host_init
            (HOST_CONTROLLER_NUMBER,   /* Use value in header file */
             MAX_FRAME_SIZE,            /* Frame size per USB spec  */
             &host_handle);
}
```

The usb_host_init() routine must be called at the interupt level. This step is necesary under OTG because a host can not be initialized at a non-interrupt level (Task level). The hardware can generate interrupts that are handled properly by the RTOS that should be handled by the host. This occurs if the execution switch is over the Task level. Freescale MQX™ Software Solutions has minimized the execution of init routines in a way that only necessary actions are done in the code implementation of this routine. This routine installs an interrupt handler for handling host interrupts like port attach, detach and so on, it also allocates the necessary memory for handling host actions.

If the software application has accepted a host or device role, it must register the driver info table to list the devices it is ready to serve and set the role as active. This is done with the following piece of code. To see the details of the driver info table, see the host API document.

```
status = _usb_host_driver_info_register
            (host_handle,
             DriverInfoTable);

  _usb_otg_set_status(otg_handle, USB_OTG_HOST_ACTIVE, TRUE);
```

Any other lines of the code must be taken as necessary actions to maintain the state of the OTG stack. They must be adopted by customer software applications with no change.

4.3 Host Applications

Assuming that the host functionality is required either because a host role has been decided under the influence of OTG state machine or a Host only USB hardware core is being used, the following steps are described to achieve the host functionality.

4.3.1 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 collection of one or more interfaces. Each interface comprises 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\device stack and describes the callback routines inside the driver info table. When a USB device is connected, the USB host\device 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 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\device stack is a few lines of code before one starts communication with the USB device. Examples on the USB stack can be written with only a host API. 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.

4.3.2 Initializing the Host Controller

The first step required to act as a host is to initialize the stack in 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 above code is the size of periodic frame list. Full speed customers can ignore the argument.

4.3.3 Register Services

Once the host is initialized, the USB Host\Device 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 on 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.

4.3.4 Enumeration Process of a Device

After the software has registered the driver info table and registered for other services, it is ready to handle devices. In the USB host\device stack, customers do not have to write any enumeration code. As soon as the device is connected to the host controller, the USB host\device stack enumerates the device and finds how many interfaces are supported. Also, for each interface it scans the registered driver info tables and finds which application has registered for the device. It provides a callback if the device criteria matches the table. The application software has to choose the interface. Here 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 */
      uint_32                          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 above code, 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`.

4.3.4.1 Selecting an Interface on 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, (pointer)&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.

4.3.4.2 Retrieving and Storing Pipe Handles

If the interface has been selected, pipe handles can be retrieved by calling as 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, pipe is a memory pointer that stores the handle (see code example for details). Notice that this routine specified the type of pipe retrieved. The code shows how to communicate to a mouse that has an interrupt pipe to obtain the pipe handle for interrupt pipe.

4.3.5 Sending/ Receiving Data to/ from Device

The USB packet transfers on USB software functions in terms of transfer requests (TR). A similar term in Windows and Linux is URB. In 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, ithe 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 exact 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);
```

4.3.6 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 shutdown the stack. These routines are obvious by their names and many are used at various places in the code. For an example, _usb_host_bus_control() routine can be used to suspend the USB bus any time under software control. Similarly, usbhost_shutdown() can be called to shut down the host stack and free all the memory. This is done when OTG applications switch roles from host to device or if the application wants to stop communicating to the USB for any reason. This routine ensures that all pipes are closed and memory freed by the stack. These functions can be used on need basis. As a suggestion search the examples that use some of these routines and copy the code if required.

4.4 Device Applications

The USB host\device stack defines enough abstraction that the same device application can work both at low-speed or full speed. However application programmers must consider detecting the speed and take appropriate actions, such as different descriptors on different speeds. A basic device application involves the following programming steps. It is a good idea to keep a source code handy to correlate with.

4.4.1 Initializing the Device Controller

Device controller initialization involves calling a device init routine in the USB host\device stack and registration of services that the device application serves. It can be noted that this example code registers one end point and bus reset callback. Whenever a transaction occurs on endpoint 0, routine service_ep0() is called. Similarly as bus reset on the USB call back reset_ep0() routine. Registration of the endpoint is not enough to start working. It is necessary to initialize and configure the endpoint using a _usb_device_init_endpoint() routine. This is best done under a reset because a reset requires the cancellation of all transfers and reinitialization of all endpoints.

```
   /* Initialize the USB device interface and receive a handle back*/
   error = _usb_device_init(0, &dev_handle, 4);

   if (error != USB_OK) {
      printf("\nUSB Device Initialization failed. Error: %x", error);
      fflush(stdout);
   } /* Endif */

   error = _usb_device_register_service(dev_handle, USB_SERVICE_EP0,
      service_ep0);

   if (error != USB_OK) {
      printf("\nUSB Device Service Registration failed. Error: %x", error);
      fflush(stdout);
   } /* Endif */

   error = _usb_device_register_service(dev_handle, USB_SERVICE_BUS_RESET,
      reset_ep0);

   if (error != USB_OK) {
      printf("\nUSB Device Service Registration failed. Error: %x", error);
      fflush(stdout);
   } /* Endif */
```

4.4.2 Prepare the Endpoint Service Routines

In USB Stack an endpoint needs initialization and configuration of hardware. Calling _usb_device_init_endpoint() routine does this. Once the endpoint is initialized, stack starts generating callbacks for the packets sent to that endpoint by the host. See the following code example:

```
_usb_device_init_endpoint(handle,                                    /* device handle */
                          1,                                         /*endpoint number */
                          max_pkt_size,                              /*max packet size 
required*/
                          USB_SED,                                   /*direction of 
endpoint,
                                                                     this is an endpoint 
that
                                                                     responds to IN*/
                          USB_BULK_ENDPOINT,                         /* type of endpoint /
                          USB_DEVICE_DONT_ZERO_TERMINATE /* default */
);
```

4.4.3 Responding to Control Endpoint Transfers

No matter what kind of application is developed, code that responds to the control end point stays the same in a USB application. To see how control end point routines respond to host enumeration requests, pickup a device example code and see how a service_ep0() routine is written.

4.4.4 Responding to a IN from Host On Non Control Endpoints

If the endpoint is initialized and the service routine is registered, call the _usb_device_send_data() routine to send data to any end point. Please see the device API document and code samples for more information. It is important to note that after a transaction is done after calling the _usb_device_send_data() routine, the USB stack generates a callback for a finished transaction. Applications can take whatever action if any action is needed for the end of a transaction.

4.4.5 Responding to an OUT From Host on Non Control Endpoints

If the endpoint is initialized and service routine is registered, call the _usb_device_recv_data() routine to receive data to any end point. Please see the device API document and code samples for more information. It is important to note that a receive data call has to be placed in advance for the stack to receive the data in the buffer provided. For an example, to expect that Bulk OUT endpoint receives data from host, queue an advance receive buffer using _usb_device_recv_data(). This can be done under a control pipe command such as set address or under a reset routine callback. Further advanced queuing must be done after endpoint receives a callback for a finished transfer. Please see the code examples with the stack for details.

Chapter 5 Debugging Applications

5.1 Introduction

Embedded development on any system comes with challenges of timing issues and unpredictable interrupt events making it necessary for programmers to dig down the source code of stack underneath. The current version of Freescale MQX™ USB Host/Device Stack provides some features for debugging the stack that can be useful for generating a real time trace of a command sent down to the stack.

Read the compilation section of this document and the release notes supplied with the stack to compile the stack in debug mode with the part of the stack (host, device, OTG or all) to debug. The following tips can be useful in debugging down the stack.

5.2 Tracing a USB Transfer in the Stack

In the current version of the stack, a real time trace can be generated at any point in application code. When this trace is enabled, all routines inside the stack layers are logged for entry and exit. This trace is useful in situations when a programmer does not know the cause of failure of a crash and wants to find the last routine called. The USB host/device stack defines a file called `host_debug.h` that defines the following macros.

- START_DEBUG_TRACE
- STOP_DEBUG_TRACE

Call these macros to start and stop tracing. For example, from `demo.c`:

```
Void main void()
{
…
…
USB initialization
START_DEBUG_TRACE /* tracing enabled */
_usb_send_data()
STOP_DEBUG_TRACE /*tracing disabled */
```

The above statements generate a trace of all routines that were entered and exited when a usb_send_data () API command was called. To see the trace, debug.h defines a global array of characters called DEBUG_TRACE_ARRAY (it is possible control the size of this array inside same file) that can be read like strings inside the debugger as a global. This array is logged by all routines when entered and exited. Some changes in the logging behavior can be made by changing the macro called DEBUG_LOG_TRACE. This macro copies the logged character string into an array. Change this to do something else, like printing the string. However, be cautious about printing because it is slow and involves several other issues with the operating system. Under many circumstances, printing from ISR routines may not be allowed and can crash the system.

Chapter 6 Key Design Issues

6.1 Timing on the USB Host\Device Stack

The USB host/device stack comes with programs that are completely OTG compliant. Customers developing OTG applications using the USB Host/Device stack have to ensure certain timing requirements imposed by the OTG compliance specification. If software does not take certain actions within a certain time, it does not pass the OTG compliance test procedure. This makes the product unable to obtain the USB OTG logo. This section lists the important timing requirements and how they are handled by the stack. The OTG specification lists several timing requirements with minimum and maximum response times to certain events.

6.1.1 Maximum Time Specified In Seconds

Any timing in seconds is not likely to meet, only if software freezes for a reason. We are not concerned about this because it is not a design issue. It is a debugging issue.

6.1.2 Maximum Time In Microseconds

Any time requirements in microseconds are not a software concern and hardware handles it.

6.1.3 Minimum Time Specified In All Cases

Any time that has minimum time constraints is not a design issue. Software can use more CPU cycles to meet requirements. In a customer design it is rare not to meet these requirements because of a technical reason, even after the software has been designed.

The table below shows the timings that software must meet.

Table 6-1. Maximum Software Times

State	Max Time	Software Action
`B_srp_init`	10 ms	When the B device wants to initiate SRP, OTG stack software pulls D+ ON and waits for a maximum of 10 ms before it pulls D+ Off. In these times a 1 ms timer interrupt from the OTG is running to ensure that D+ is off before 10 ms. Customer application must ensure that the 1 ms timer is not interrupted by any other higher priority interrupts when the SRP is running. If it needs to be interrupted, the execution time should be < 5 ms to provide for 5 ms for the USB-OTG peripheral.
`B_srp_init`	100ms	When B device generates SRP, software waits for 100 ms maximum to find if the host turns on the VBUS. If the host does not respond within 100 ms, the B device assumes that SRP has failed. OTG software generates a device callback to inform the device application that the SRP attempt has failed.

Table 6-1. Maximum Software Times

`b_peripheral`	150ms	In USB Host/Device stack, HNP by a B_DEVICE is an application-initiated event therefore the stack does not time when the bus is idle. This timer is not implemented by the stack.
`B_wait_acon`	1 ms	This is the hardest timer imposed by OTG compliance. When the B device wants to become a B_HOST, it must reset the bus within 1 ms of detecting a connection by the A device. When the A device performs a connection. the USB host/device OTG software changes roles from B device to B Host. The change of role includes the shutdown of the B device stack and the start of host stack.
`A_wait_vrise`	100ms	When A device is in `A_wait_vrise`, it waits for 100 ms before moving to `A-wait_bcon`. If a customer implementation causes the USB host/device stack timer to be preempted (causing the real wait to be more than 100 ms), it does not cause any harm in general. This timer can be reduced to take less time and therefore a flexible parameter in implementation.
`A_suspend`	3ms	Under HNP, when device is in a suspend state and B Device does a disconnect, it means that it wants to become host. This time must be less then 3 ms.
`A peripheral`	200ms	In USB host/device stack `A peripheral time` is controlled by the application because the application can decide if it does not want to act like a peripheral because of lack of activity on the bus. The time measured in the current applications are about 3–4 ms and can be adjusted by putting a delay in the application. This time must not cause concern to software developers.
`A_wait_bcon`	100ms	This is a short debounce interval and is not tested by the OTG compliance procedure. However, USB host/device stack allows debounce interval to be controlled.

6.2 Ensuring an OTG Compliant Product

To ensure that the product is compliant OTG run the OPT tester on the product. This tester generates log files that can be compared with the log files released with the USB host/device stack to find new issues caused and to point precisely on compliance failures.

6.2.1 Actions Taken if a Product Failures

By nature of embedded systems, software timings are implementation dependent and therefore there is no best way of arranging a piece of code. However, the USB host/device stack is flexible enough to gain and reduce times at various places to meet requirements.

6.3 Software Memory Requirements

Another important issue from customer perspective is the code and data memory requirement of USB host/device stack. When an example is compiled with Freescale MQX™ Design and Development Tools, a .xmap file is generated that defines the size and location of each routine. This file can be used to measure the code memory requirement of the stack. Stack size also depends upon the class driver libraries used. Ensure that compiled example is for the class needed. The following table can help understand data memory requirements.

Table 6-2. USB Host/Device Stack Memory Requirements

Memory Requirement	Who allocates it?	How much?	USB HID (Mouse) Example for MCF52259
Data buffers to send or receive data + any application dependent storage	Software application written by customer	Depends upon the customer application	Global variables: 20 bytes Interrupt pipe buffer: 4 Bytes Total: 24 Bytes
USB system management data structures such as pipe handler, device handler, link list of transfers, and host and device state structures	USB host\device stack middle layer (Common class API)	It is a dynamic memory allocation that depends upon the number of devices on the system, number of endpoints, size of descriptors, and nature of device (bulk only or ISO device) and so on.	Endpoint pipes: 144 Bytes Device instance information: 148 Bytes Host instance information: 732 Bytes Total: 1024 Bytes
USB hardware management data structures	USB hardware drivers (KHCI and DCI drivers)	This memory is hardware core dependent. Most of this memory is allocated once when the stack is initialized by a routines like `usb_host_init()` and so on.	Global variables: 492 Bytes KHCI task stack: ~1.4kB top (worst case) Total: ~2 kB