i.MX23 EVK Windows Embedded CE 6.0

Reference Manual

Document Number: 924-76402
Rev. 2009.12

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

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

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

Contents

About This Book

Audience xi
Suggested Reading xi
Conventions xi
Definitions, Acronyms, and Abbreviations xi

Chapter 1 Introduction

1.1 Getting Started 1-1
1.2 Windows Embedded CE 6.0 Architecture 1-1

Chapter 2 Audio Driver

2.1 Audio Driver Summary 2-1
2.2 Supported Functionality 2-2
2.3 Hardware Operation 2-2
2.3.1 Audio Hardware Design 2-2
2.3.2 Audio Playback 2-2
2.3.3 Audio Recording 2-3
2.3.4 Required SoC Peripherals 2-3
2.3.5 Conflicts with SoC Peripherals 2-3
2.3.6 Conflicts with Board Peripherals 2-3
2.3.7 Known Issues 2-3
2.4 Software Operation 2-4
2.4.1 Audio Playback 2-4
2.4.2 Audio Recording 2-4
2.4.3 Audio Driver Compile-Time Configuration Options 2-4
2.4.4 DMA Support 2-5
2.4.5 Power Management 2-6
2.4.6 Audio Driver Registry Settings 2-7
2.5 Unit Test 2-8
2.5.1 Unit Test Hardware 2-8
2.5.2 Unit Test Software 2-8
2.5.3 Building the Audio Driver CETK Tests 2-9
2.5.4 Running the Audio Driver CETK Tests 2-9
2.6 System Level Audio Driver Tests 2-9

2.6.1 Checking for a Boot-Time Musical Tune 2-9
2.6.2 Confirming Touchpanel Taps and Keypad Key Presses 2-9
2.6.3 Playing Back Sample Audio and Video Files Using the Media Player 2-9
2.6.4 Using the SDK Sample Audio Applications for Testing 2-10
2.7 Audio Driver API Reference 2-10
2.8 Audio Driver Troubleshooting Guide 2-10
2.8.1 Checking Build-Time Configuration Options 2-10
2.8.2 Media Player Application Not Found 2-10
2.8.3 Media Player Fails to Load and Play an Audio File 2-10

Chapter 3
Backlight Driver

3.1 Backlight Driver Summary 3-1
3.2 Supported Functionality 3-1
3.3 Hardware Operation 3-1
3.3.1 i.MX233-EVK Hardware Operation 3-2
3.4 Software Operation 3-2
3.4.1 Backlight Driver Registry Settings 3-2
3.4.2 Power Management 3-2
3.5 Unit Test 3-3
3.5.1 Unit Test Hardware 3-3
3.5.2 Unit Test Software 3-3
3.5.3 Running the Backlight Application Test 3-4
3.6 Backlight API Reference 3-4

Chapter 4
Battery Driver

4.1 Battery Driver Summary 4-1
4.2 Supported Functionality 4-1
4.3 Hardware Operation 4-2
4.3.1 Conflicts with Other SoC Peripherals 4-2
4.4 Software Operation 4-2
4.4.1 Battery Driver Registry Settings 4-2
4.4.2 Power Management 4-2
4.5 Unit Test 4-2
4.5.1 Unit Test Hardware 4-3
4.6 Battery API Reference 4-3

Chapter 5
Boot from Secure Digital/MultiMedia Card (SD/MMC)

5.1 Boot from SD/MMC Summary 5-1
5.2 Supported Functionality 5-1
5.3 Hardware Operation 5-2

5.3.1 Conflicts with Other Peripherals and Catalog Items 5-2
5.4 Software Operation 5-2
5.5 Card Flashing Tool 5-2
5.5.1 Write Image (EBOOT) to SD Card 5-2
5.5.2 System Boot 5-2

Chapter 6
Chip Support Package Driver Development Kit (CSPDDK)

6.1 CSPDDK Driver Summary 6-1
6.2 Supported Functionality 6-1
6.3 Hardware Operation 6-2
6.3.1 Conflicts with Other Peripherals and Catalog Items 6-2
6.4 Software Operation 6-2
6.4.1 Communicating with the CSPDDK 6-2
6.4.2 Compile-Time Configuration Options 6-2
6.4.3 Registry Settings 6-2
6.4.4 Power Management 6-3
6.5 Unit Test 6-3
6.5.1 CSPDDK DLL System Clocking (DDK_CLK) Reference 6-3
6.5.2 CSPDDK DLL GPIO (DDK_GPIO) Reference 6-6
6.5.3 CSPDDK DLL IOMUX (DDK_IOMUX) Reference 6-8
6.5.4 CSPDDK DLL DMA (DDK_DMA) Reference 6-10
6.5.5 CSPDDK POWER (DDK_POWER) Reference 6-15

Chapter 7
Configurable Serial Peripheral Interface (CSPI) Driver

7.1 CSPI Driver Summary 7-1
7.2 Supported Functionality 7-1
7.2.1 Conflicts with Other Peripherals and Catalog Items 7-1
7.2.2 Conflicts with EVK Peripherals 7-2
7.3 Software Operation 7-2
7.3.1 Registry Settings 7-2
7.3.2 Communicating with the CSPI 7-2
7.3.3 Creating a Handle to the CSPI 7-2
7.3.4 Data Transfer Operations 7-3
7.3.5 Closing the Handle to the CSPI 7-4
7.3.6 Power Management 7-4
7.4 Unit Test 7-5
7.4.1 Building the Unit Tests 7-5
7.5 CSPI Driver API Reference 7-5
7.5.1 CSPI Driver IOCTLs 7-6
7.5.2 CSPI Driver SDK Wrapper 7-6
7.5.3 CSPI Driver Structures 7-7

Chapter 8
Display Driver for LCDIF and PXP

8.1 Display Driver Summary 8-1
8.2 Supported Functionality 8-1
8.3 Hardware Operation 8-2
8.3.1 Conflicts with Other Peripherals and Catalog Items 8-2
8.4 Software Operation 8-2
8.4.1 Software Driver Components 8-3
8.4.2 Communicating with the Display 8-4
8.4.3 Configuring the Display 8-5
8.4.4 Power Management 8-5
8.5 Unit Test 8-6
8.5.1 Unit Test Hardware 8-6
8.5.2 Unit Test Software 8-6
8.5.3 Building the Unit Tests 8-7
8.5.4 Running the Unit Tests 8-8
8.6 Display Driver API Reference 8-8

Chapter 9
Dynamic Voltage and Frequency Control (DVFC) Driver

9.1 DVFC Driver Summary 9-1
9.2 Supported Functionality 9-1
9.2.1 i.MX233 Supported Functionality 9-2
9.3 Hardware Operation 9-2
9.3.1 Conflicts with Other Peripherals and Catalog Items 9-2
9.3.2 i.MX233 EVK Configuration 9-2
9.4 Software Operation 9-2
9.4.1 i.MX233 Registry Settings 9-2
9.4.2 Loading and Initialization 9-2
9.4.3 Operation 9-2
9.4.4 DDK Interface 9-3
9.4.5 Power Management 9-3
9.5 Unit Test 9-4
9.5.1 i.MX233 Unit Testing 9-4

Chapter 10
Keypad Driver

10.1 Keypad Driver Summary 10-1
10.2 Supported Functionality 10-1
10.3 Hardware Operation 10-1
10.3.1 Conflicts with Other Peripherals and Catalog Items 10-1
10.3.2 Keypad 10-2
10.4 Software Operation 10-2

10.4.1 Keypad Scan Codes and Virtual Keys 10-2
10.4.2 Power Management 10-3
10.4.3 Keypad Registry Settings 10-3
10.5 Unit Test 10-4
10.5.1 Unit Test Hardware 10-4
10.5.2 Unit Test Software 10-4
10.5.3 Building the Unit Tests 10-4
10.5.4 Running the Unit Tests 10-4

Chapter 11
Inter-Integrated Circuit (I^2C) Driver

11.1 I^2C Driver Summary 11-1
11.2 Supported Functionality 11-1
11.3 Hardware Operation 11-1
11.3.1 Conflicts with Other Peripherals and Catalog Items 11-2
11.4 Software Operation 11-2
11.4.1 Registry Settings 11-2
11.4.2 Communicating with the I^2C 11-2
11.4.3 Creating a Handle 11-3
11.4.4 Configuring the I^2C 11-3
11.4.5 Data Transfer Operations 11-4
11.4.6 Closing the Handle 11-5
11.5 Unit Test 11-5
11.5.1 Unit Test Hardware 11-5
11.5.2 Unit Test Software 11-6
11.5.3 Building the Unit Tests 11-6
11.5.4 Running the Unit Tests 11-6
11.6 Hardware Limitations 11-6
11.7 I^2C Driver API Reference 11-6
11.7.1 I^2C Driver IOCTLS 11-7
11.7.2 I^2C Driver SDK Encapsulation 11-9
11.7.3 I^2C Driver Structures 11-12

Chapter 12
Low-Resolution Analog-Digital Converter (LRADC) Driver

12.1 LRADC Driver Summary 12-1
12.2 Supported Functionality 12-1
12.3 Hardware Operation 12-2
12.3.1 Conflicts with Other Peripherals and Catalog Items 12-2
12.4 Software Operation 12-2
12.4.1 ADC Registry Settings 12-2
12.4.2 Interfacing with the LRADC Driver 12-2
12.5 Power Management 12-2
12.5.1 LDC_PowerUp 12-3

12.5.2 LDC_PowerDown 12-3
12.5.3 IOCTL_POWER_CAPABILITES 12-3
12.5.4 IOCTL_POWER_SET 12-3
12.5.5 IOCTL_POWER_GET 12-3
12.6 Unit Test 12-3
12.7 LRADC SDK API Reference 12-3
12.7.1 LRADC SDK Functions 12-3

Chapter 13
NAND Flash Driver

13.1 Flash Driver Summary 13-1
13.2 Supported Functionality 13-2
13.3 Hardware Operation 13-2
13.3.1 Conflicts with Other Peripherals and Catalog Items 13-2
13.4 Software Operation 13-2
13.4.1 MDD and PDD Layer Overview 13-2
13.4.2 Data Structures 13-4
13.4.3 Adding New Flash Configurations 13-6
13.4.4 Registry Settings 13-7
13.4.5 DMA Support 13-7
13.4.6 Power Management 13-7
13.5 Unit Test 13-7
13.5.1 CETK Testing 13-7
13.5.2 System Testing 13-8

Chapter 14
NAND Redundant Boot

14.1 NAND Redundant Boot Summary 14-1
14.2 Supported Functionality 14-1
14.3 Hardware Operation 14-2
14.3.1 Conflicts with Other Peripherals and Catalog Items 14-2
14.4 Software Operation 14-2
14.5 Unit Test 14-4
14.5.1 Testing Update Functionality 14-4
14.5.2 Testing Restore Functionality 14-4

Chapter 15
Serial Driver

15.1 Serial Driver Summary 15-1
15.2 Supported Functionality 15-1
15.3 Hardware Operation 15-2
15.3.1 Conflicts with Other Peripherals and Catalog Items 15-2
15.3.2 Known Issues 15-2

15.4 Software Operation 15-2
15.4.1 Registry Settings 15-2
15.4.2 Power Management 15-3
15.5 Unit Test 15-3
15.5.1 Unit Test Hardware 15-3
15.5.2 Unit Test Software 15-3
15.5.3 Building the Unit Tests 15-3
15.5.4 Running the Unit Tests 15-4
15.6 Serial Driver API Reference 15-4
15.6.1 Serial PDD Functions 15-5
15.6.2 Serial Driver Structures 15-6

Chapter 16
Secure Digital Host Controller (SDHC) Driver

16.1 SDHC Driver Summary 16-1
16.2 Supported Functionality 16-1
16.3 Hardware Operation 16-2
16.3.1 Conflicts with Other Peripherals and Catalog Options 16-2
16.4 Software Operation 16-2
16.4.1 Required Catalog Items 16-2
16.4.2 SDHC Registry Settings 16-3
16.4.3 DMA Support 16-3
16.4.4 Power Management 16-3
16.5 Unit Test 16-3
16.5.1 Unit Test Hardware 16-4
16.5.2 Unit Test Software 16-4
16.5.3 Building the Unit Tests 16-4
16.5.4 Running the Unit Tests 16-4
16.5.5 System Testing 16-6
16.6 Secure Digital Card Driver API Reference 16-6

Chapter 17
Touch Panel Driver

17.1 Touch Panel Driver Summary 17-1
17.2 Supported Functionality 17-1
17.3 Hardware Operations 17-1
17.4 Software Operations 17-2
17.4.1 Touch Driver Registry Settings 17-2
17.5 Unit Tests 17-3
17.5.1 Unit Test Hardware 17-3
17.5.2 Unit Test Software 17-3
17.5.3 Running the Touch Panel Tests 17-4
17.6 Touch Panel API Reference 17-4

Chapter 18
Universal Serial Bus (USB) On The Go (OTG) Driver

18.1 USB OTG Driver Summary 18-1
18.1.1 Peripheral Driver Summary 18-1
18.1.2 Host Driver Summary 18-2
18.1.3 OTG (Pin-Detection) Driver Summary 18-3
18.2 Supported Functionality 18-3
18.3 Hardware Operation 18-4
18.3.1 Conflicts with Other Peripherals and Catalog Items 18-4
18.4 Software Operation 18-4
18.4.1 USB OTG Host Controller Driver 18-4
18.4.2 USB Peripheral Driver 18-12
18.4.3 USB OTG Driver (Pin-Detection Driver) 18-16
18.4.4 USB OTG Catalog Settings 18-18
18.4.5 USB OTG Registry Settings 18-18
18.4.6 Power Management 18-20
18.4.7 Peripheral Class Drivers 18-23
18.4.8 Host Class Drivers 18-26
18.5 Known Issues 18-28
18.5.1 Host Support for Low Speed Peripherals 18-28
18.5.2 Host VBUS Power Supply 18-28
18.6 Basic Elements for Driver Development 18-28
18.6.1 BSP Environment Variables 18-28
18.6.2 Dependencies of Drivers 18-29
18.7 Application Tools for USB 18-29
18.7.1 Application for USB Peripheral Class Driver Switch 18-29
18.7.2 Application for Multispec PHDC Demo 18-30

Chapter 19
USB Boot and KITL

19.1 USB Boot and KITL Summary 19-1
19.2 Supported Functionality 19-1
19.3 Hardware Operation 19-1
19.3.1 Conflicts with Other Peripherals and Catalog Items 19-2
19.4 Software Operation 19-2
19.4.1 Software Architecture 19-2
19.4.2 Source Code Layout 19-3
19.4.3 Power Management 19-3
19.4.4 Registry Settings 19-3
19.4.5 DMA Support 19-3
19.5 Unit Test 19-3
19.5.1 Building the USB Boot and KITL 19-4

About This Book

This reference manual describes the requirements, implementation details, and testing for each module included in the Freescale software development kit (SDK) for Microsoft® Windows® CE 6.0.

Audience

This document is intended for device driver developers, application developers, and software test engineers who are planning to use the product. This document is also intended for people who want to know more about Freescale's software development kit (SDK) for Microsoft Windows CE 6.0.

Suggested Reading

The Freescale manuals can be found at the Freescale Semiconductor, Inc. World Wide Web site listed on the front cover of this document. These manuals can be downloaded directly from the Web site, or printed versions can be ordered. The Microsoft Platform Builder Help may be viewed from within the Platform Builder application.

- i.MX233 Applications Processor IC Reference Manual
- i.MX233 Release Notes for Windows Embedded CE 6.0
- i.MX233 User's Guide for Windows Embedded CE 6.0
- Microsoft Platform Builder for Windows Embedded CE 6.0 Help

Conventions

This document uses the following notational conventions:

- `Courier` indicates directory or file names and code examples.
- **Bold** indicates the menu options or buttons the user can select. Cascaded menu options are delimited with the > symbol.
- *Italic* indicates a reference to another document.

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

API	application programming interface
BSP	board support package
CSP	chip support package
CSPI	configurable serial peripheral interface
D3DM	Direct 3D Mobile
DHCP	dynamic host configuration protocol
DPTC	dynamic power and temperature control
DVFC	dynamic voltage and frequency control
DVFS	dynamic voltage and frequency scaling
EBOOT	Ethernet bootloader

EVB	platform evaluation board
FAL	flash abstraction layer
FIR	fast infrared
FMD	flash media driver
GDI	graphics display interface
GPT	general purpose timer
I^2C	inter-integrated circuit
IDE	integrated development environment
IST	interrupt service thread
IPU	image processing unit
KITL	kernel independent transport layer
LVDS	low-voltage differential signaling
MAC	media access control
MMC	multimedia cards
OAL	OEM adaptation layer
OEM	original equipment manufacturer
OS	operating system
OTG	on-the-go
PMIC	power management IC
PQOAL	production quality OEM adaptation layer
PWM	pulse-width modulator
SD	secure digital cards
SDC	synchronous display controller
SDHC	secure digital host controller
SDIO	secure digital I/O and combo cards
SDRAM	synchronous dynamic random access memory
SDK	software development kit
SIM	subscriber identification module
SOC	system on a chip
UART	universal asynchronous receiver transmitter
USB	universal serial bus

Chapter 1
Introduction

This Freescale board support package (BSP) is based on the Microsoft Windows® Embedded CE 6.0 operating system. This BSP supports the following Freescale platform(s):

- i.MX233-EVK Development System

This kit supports the Microsoft Windows Embedded CE 6.0 operating system, and requires the use of the Microsoft Platform Builder, which is an integrated development environment (IDE) for building customized embedded OS designs. To view feature information, see the *BSP Release Notes*.

NOTE

Use this guide in conjunction with the Microsoft Windows Platform Builder Help (or the identical *Platform Builder User Guide)*.

- To view the Platform Builder Help, click **Help** from the Platform Builder application.
- To view the online Windows Embedded CE 6.0 documentation, visit: http://msdn2.microsoft.com/en-us/library/bb159115.aspx

1.1 Getting Started

For instructions on installing this software release, building, downloading and running the OS image on the hardware board, see the appropriate User's Guide.

1.2 Windows Embedded CE 6.0 Architecture

The Windows Embedded CE 6.0 architecture is a variation of the Windows OS for minimalistic computers and embedded systems. The architecture of the operating system and sub-systems (for example, power management or DirectDraw) are described in several locations in the Help. Begin at the following location in Help topic:

Welcome to Windows Embedded CE 6.0 > Windows Embedded CE Architecture

Chapter 2
Audio Driver

The audio driver module provides audio playback and recording functions. For information about accessing an application with the audio driver using the methods and functions associated with the WaveOut or WaveIn functionality, see the Platform Builder Help at the following location:

Windows Embedded CE Features > Audio > Waveform Audio > Waveform Audio Application Development

2.1 Audio Driver Summary

Table 2-1 provides the source code location, library dependencies, and other BSP information.

Table 2-1. Audio Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\WAVEDEV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\WAVEDEV2
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\WAVEDEV2
Driver DLL	wavedev2_mx233.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX233 EVK:ARMV4I > Device Drivers > Audio > Audio driver
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO=

NOTE

The selection and use of the Windows Media Player and the various software codecs is beyond the scope of the audio driver and is not discussed in this document. For information about these items, see the Platform Builder Help at the following location: **Windows Embedded CE Features > Audio**

2.2 Supported Functionality

The audio driver enables the system to provide the following software and hardware support:

1. Conforms to the audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Double-buffered DMA operations to transfer audio data between memory and the hardware FIFO
3. Two power management modes: full on and full off
4. Full duplex playback and record
5. Minimizes power consumption at all times by using clock gating and by disabling all audio-related hardware components that are not actively being used
6. 8–96 KHz for both recording and playback
7. Mono and stereo 16-bit sample

2.3 Hardware Operation

This section describes about the audio hardware operation.

2.3.1 Audio Hardware Design

This section describes the connection between the SoC audio peripherals and the external audio codec, the access interface of audio codec, and the audio input or output device connections.

2.3.1.1 i.MX233 Audio Hardware Design

The i.MX233 includes the AUDIOOUT/DAC and AUDIOIN/ADC modules on chip. PCM audio samples are transferred from a buffer in memory to the AUDIOOUT FIFO, and output to the analog DAC. The analog audio destination can be stereo headphone amplifier or speaker amplifier on board. The i.MX233 also features an audio record path that consists of a sigma-delta analog-to-digital converter (ADC), followed by the AUDIOIN digital multi-stage FIR filter. The ADC oversamples the input from the microphone or line-in on board.

2.3.2 Audio Playback

By default, the following hardware configuration options are enabled for the playback operation (based on the default audio driver configuration):

- The audio driver is configured to use AUDIOOUT/DAC module and a sample rate at 44.1 KHz.
- The APBX DMA channel is setup to support data between application memory buffers and the AUDIOOUT FIFO.
- The DAC is enabled to begin the transmission of audio data stream.
- An interrupt is generated when a DMA buffer is empty and this interrupt is handled by the audio driver. The audio driver refills the DMA buffer and returns it to the DMA controller for processing.
- Due to the double-buffering scheme, the DMA controller simply uses the other DMA buffer to continue refilling the AUDIOOUT FIFO while the previous DMA buffer is being refilled.

2.3.3 Audio Recording

The following hardware configuration steps are performed just prior to each recording operation (based upon the default audio driver configuration):

- The audio driver is configured to use AUDIOIN/ADC module for recording path.
- The APBX DMA channel is setup to support data between application memory buffers and the AUDIOIN FIFO.
- The ADC is enabled to begin the receiver of audio data stream.
- An interrupt is generated when a DMA buffer is full and this interrupt is handled by the audio driver. The audio driver copies the data from the full input DMA buffer into application-supplied buffers and returns the empty DMA buffer to the DMA controller for processing.
- Due to the double-buffering scheme, the DMA controller simply uses the other DMA buffer to continue receiving data from the AUDIOIN FIFO while the previous DMA buffer is being copied.

2.3.4 Required SoC Peripherals

Table 2-2 shows the SoC hardware components required by the audio driver.

Table 2-2. Required SoC Peripherals

Component	Use
AUDIOOUT/DAC	Playback
AUDIOIN/ADC	Record
APBX DMAr	Manages the DMA channels that are used for playback and recording

2.3.5 Conflicts with SoC Peripherals

No conflicts.

2.3.6 Conflicts with Board Peripherals

The following section explains about the conflicts of the audio driver with board peripherals:

2.3.6.1 i.MX233 Peripherals Conflicts

No conflicts.

2.3.7 Known Issues

The following section explains about the known issues in the audio driver:

2.3.7.1 i.MX233 Known Issues

If coexistence of stereo audio driver and S/PDIF driver occurs, the default audio device might be S/PDIF. The default audio device may be chosen by AudioRouting application.

2.4 Software Operation

The audio driver follows the Microsoft-recommended architecture for audio drivers. For information about the architecture and operation, see the Platform Builder Help at the following location:

Developing a Device Driver > Windows CE Drivers > Audio Drivers > Audio Driver Development Concepts

2.4.1 Audio Playback

The software operation of the audio driver for playback is similar to the hardware configuration. Once the hardware components are configured, the audio driver only handles the output DMA buffer empty interrupts. This is done by the interrupt handler, which refills each of the output DMA buffers with new audio data that has been supplied by the application, and then returns the DMA buffer to the DMA controller.

2.4.2 Audio Recording

The operation of the audio driver for recording is similar to the hardware configuration. Once the hardware components are configured, then the audio driver handles the input DMA buffer full interrupts. This is done by the interrupt handler, which copies the contents of each input DMA buffer to an application-supplied buffer, and then returns the empty DMA buffer to the DMA controller. If the application-supplied buffer does not have enough space for all of the new data, discard any extra data. The application is signaled using a callback function when the application-supplied buffer is full.

2.4.3 Audio Driver Compile-Time Configuration Options

The audio driver can be configured for a wide variety of operating modes depending on the hardware and software requirements.

NOTE

Do not change the audio driver configuration settings without a detailed understanding of the platform hardware configuration and operating characteristics. Selecting invalid or incorrect configuration settings may result in the audio driver not loading or operating properly. Conversely, the audio driver performance and resource usage may be fine-tune by adjusting these configuration settings. For further information about the configuration options, see the corresponding source files.

2.4.3.1 i.MX233 Audio Driver Configuration Options

Table 2-3 gives the compile-time configuration options of the i.MX233 stereo audio driver.

Table 2-3. i.MX233 Audio Driver Configuration Options (oemsettings.h)

Configuration Setting Name	Description
INCHANNELS	Defines the number of input/recording channels that are available. Can be set to either 1 or 2. Default is 2.
OUTCHANNELS	Defines the number of output/playback channels that are available. Can be set to either 1 or 2. Default is 2.
BITSPERSAMPLE	The number of data bits per audio sample. This must match with the HWSAMPLE typedef and the AUDIO_SAMPLE_MAX/AUDIO_SAMPLE_MIN values. Default is 16.
HWSAMPLE	A typedef that defines the size of each audio data word. This must match the BITSPERSAMPLE and AUDIO_SAMPLE_MAX/AUDIO_SAMPLE_MIN values. Default is 16.
USE_MIX_SATURATE	Enable a check in the software mixer code to guard against saturation. Default is 1.
AUDIO_SAMPLE_MAX and AUDIO_SAMPLE_MIN	The valid range of each audio data word. Values that are outside of this range is clipped to the max/min value by the saturation protection code if USE_MIX_SATURATE is set to 1. Default is 32767 and -32768.
ENABLE_MIDI	If set to 1, MIDI code is included in the driver (~4 Kbytes).
USE_OS_MIXER	If set to 1, the driver does not do any internal mixing and relies on the OS mixer.

2.4.4 DMA Support

The audio driver uses the DMA controller to transfer digital audio data between the audio application and the audio FIFOs. This minimizes the processing required by the ARM core and can also reduce the power consumption during audio playback and recording operations. This section describes the audio driver DMA implementation issues and trade-offs, and the available compile-time DMA-related configuration options.

To use DMA transfers, the following items must be properly allocated, managed, and deallocated by the device driver:

- The DMA data buffers where the application data is kept
- The DMA buffer descriptors, which are used by the DMA hardware to manage the state of each DMA buffer

The DMA data buffers can be allocated from either the internal memory (which is provided by on-chip internal RAM) or external memory (which is provided by off-chip external DRAM).

Table 2-4 describes the issues and considerations for the type of memory to use for the DMA data buffers.

Table 2-4. DMA Memory Allocation Issues and Considerations

Memory Region	Memory Usage Issues and Considerations
Internal	• Allows the external memory to be placed in a low power mode while the DMA data buffers are being processed to reduce system power consumption (as long as nothing else on the system requires access to external memory) • Less power is required to access the internal RAM • The total size of the internal memory region is limited • The limited amount of internal memory may have to be shared by multiple device drivers • The entire internal memory region must be manually managed with predefined addressed ranges being reserved for each specific use
External	• The total size of the external memory is typically much greater than the size of the internal memory. This provides much greater flexibility in selecting the size of the DMA data buffers. • There is typically no need to worry about the possible impact and memory requirements of any other device driver. • Memory allocation is handled using the standard Windows Embedded CE 6.0 system calls • The external memory cannot be placed into a low power mode while the DMA is active

Table 2-5 describes how to configure the build so that the audio driver allocates its DMA data buffers from either the internal or external memory. The DMA buffer descriptors can also be allocated either from internal or external memory.

Table 2-5. Configuration Options for Internal or External Memory DMA Data Buffer Allocation

Memory Region	Required Configuration Options
Internal	Set the BSP_AUDIO_DMA_BUF_ADDR macro in bsp_cfg.h to an address within the internal memory region. Set BSP_AUDIO_DMA_BUF_SIZE to the total size (in bytes) for all DMA data buffers that is allocated.
External	Make sure that the BSP_AUDIO_DMA_BUF_ADDR macro is commented out in bsp_cfg.h

2.4.4.1 i.MX233 Audio DMA Buffer Use

The i.MX233 audio driver supports both playback and recording. Both playback function and recording function allocate DMA buffer from external memory.

2.4.5 Power Management

The primary method for limiting power consumption in the audio driver is to gate off all clocks to the SSI when those clocks are not needed, and to turn off all audio hardware components at the end of each audio stream. This is accomplished through the **DDKClockSetGatingMode** function call and the various PMIC audio APIs. In the BSP, the audio module can be disabled, and its clocks are turned off whenever there are no active audio I/O operations. The clock gating and the disabling of related audio hardware components is handled automatically within the audio module and requires no additional configuration or code changes.

The audio driver operates correctly when resuming after the power down mode.

2.4.5.1 PowerUp

This function resumes an audio I/O operation that was previously terminated by calling the PowerDown() API. It begins by restoring power and re-enabling all of the required audio hardware components. Then this function restarts the audio DMA transfers to complete the powerup process for the audio driver.

This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. Therefore, all required timed delays must be handled by using a polling loop instead of any of the normal **wait for an event to be signalled** functions. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

2.4.5.2 PowerDown

This function suspends all currently active audio I/O operations just before the entire system enters the low power state. This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. So, first thing this function must do is to signal all of the possible wait events that the normal audio driver thread may currently be waiting on. If this function does not signal all waiting events, the PowerDown thread may be blocked waiting for a critical section that is currently being held by the normal audio driver thread. This deadlocks the entire system and prevent it from properly entering the low power state.

When all waiting events are signalled, the normal audio thread is guaranteed (because of priority inversion) to run to the point where it releases the required critical section and allows the PowerDown thread to proceed without the possibility of deadlocking.

When the normal audio thread is not executing inside any critical section, the PowerDown thread can safely proceed to disable all active audio DMA operations and to power down the associated audio hardware components. Once this is done, the audio driver remains in a low power state until the PowerUp function is called by the Power Manager. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

2.4.5.3 IOCTL_POWER_SET

This Power Manager IOCTL is implemented for the audio driver. All system suspend and resume functions are handled by the IOCTL, which manages the PowerDown and PowerUp functionality. For all platforms, the following registry entry must be defined:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
        "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
```

This registry entry is required for proper power management functionality.

2.4.6 Audio Driver Registry Settings

At least one registry key must be properly defined so that the Device Manager loads the audio driver when the system is booted. Additional registry keys may also be defined and changed at runtime, to configure the operation of the audio driver.

2.4.6.1 i.MX233 Audio Driver Registry Settings

The following registry keys are required for the Device Manager to properly load the i.MX233 audio device driver during the device normal boot process. These registry settings should not be modified. If the settings are missing or incorrectly defined, then the audio driver may not be loaded and all audio functions are disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
    "Prefix"="WAV"
    "Dll"="wavedev2_mx233.dll"
    "Index"=dword:2
    "Order"=dword:4
    "Priority256"=dword:95
    "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}",
                      "{37168569-61C4-45fd-BD54-9442C7DBA46F}"
; Override wave API load order to follow audio driver
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN]
   "Order"=dword:5
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN_ACM]
   "Order"=dword:5
```

2.5 Unit Test

The audio driver is tested using the Waveform Audio Driver Test suite included with the Windows Embedded CE 6.0 Test Kit (CETK). The test suite includes automated and interactive tests used to test playback and recording functions.

2.5.1 Unit Test Hardware

Table 2-6 identifies the hardware needed to run the unit tests.

Table 2-6. Hardware Requirements

Requirement	Description
Stereo headphones or earphones	This is required to confirm that audio playback is working. The headphones or earphones should have a 3.5 mm jack
Mono microphone	—

2.5.2 Unit Test Software

Table 2-7 lists the software required to run the unit tests.

Table 2-7. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
wavetest.dll	Test.dll file

2.5.3 Building the Audio Driver CETK Tests

The audio driver tests come pre-built as part of the CETK. No steps are required to build these tests. The wavetest.dll file is included with the CETK files in the following location:

```
[Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I
```

2.5.4 Running the Audio Driver CETK Tests

The command line for running the audio driver test is:

```
tux –o –d wavetest
```

Alternatively, use the CETK interface in the Platform Builder. If the full-duplex operation is not supported, the command line is:

```
tux -o -d wavetest -c “-e”
```

For detailed information about the audio driver tests, see the Platform Builder Help at the following location:

Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Audio Tests > Waveform Audio Driver Test

2.6 System Level Audio Driver Tests

In addition to running the audio driver tests in the CETK, various system-level tests that involve the use of the audio driver can be performed. The following sections describe how to test the audio driver without using the CETK.

2.6.1 Checking for a Boot-Time Musical Tune

The normal Windows Embedded CE 6.0 boot procedure includes playing a short musical tune just before displaying the touch panel calibration screen. At this point, the audio driver should already have successfully loaded and the tune should be heard if a headset is attached to the stereo output jack.

2.6.2 Confirming Touchpanel Taps and Keypad Key Presses

The normal Windows Embedded CE 6.0 system configuration includes the ability to playback a short tapping sound when the stylus makes contact with the touchpanel. These taps should be heard when a headset is attached to the stereo output jack. A click should also be heard when a key on the keypad is pressed.

2.6.3 Playing Back Sample Audio and Video Files Using the Media Player

The Microsoft-supplied Media Player application can be used to load and play a variety of audio and video media files in a number of different formats. The only requirement is to include the software codecs in the OS image that may be needed to decode the media file. The Media Player includes controls for pausing, resuming, and stopping playback, and advancing playback to a specific point. Volume and muting controls are also provided.

2.6.4 Using the SDK Sample Audio Applications for Testing

The Windows Embedded CE 6.0 SDK that is included as part of the Platform Builder includes two audio-related sample applications. The `wavrec` sample application can be used to test the audio recording function while the `wavplay` sample application provides a command line-based method of playing back various media files. For additional information about these sample applications, see the Platform Builder Help at the following location:

Windows Embedded CE Features > Audio > Waveform Audio > Waveform Audio Samples

2.7 Audio Driver API Reference

For detailed reference information for the audio driver, see the Platform Builder Help at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Audio Drivers > Audio Driver Reference > Waveform Audio Driver Reference

2.8 Audio Driver Troubleshooting Guide

This section describes the techniques to identify and fix the most common problems involving the audio driver.

2.8.1 Checking Build-Time Configuration Options

Compile-time or link-time errors are probably occur due to incorrect or invalid configuration settings defined in `hwctxt.h` or `hwctxt.cpp`. See Section 2.4.3.1, "i.MX233 Audio Driver Configuration Options," for information about the device driver build configuration options. Follow the build procedure documented in the Release Notes to compile and link the audio driver. Confirm that the required Platform Builder catalog items are included in the OS design. See Table 2-1 for a list of the required and recommended audio driver-related catalog items.

2.8.2 Media Player Application Not Found

Make sure that the Media Player catalog item is included in the OS design. The Media Player application is not included in the final system image if the catalog item is not selected. For more information on this topic, see the Platform Builder Help at the following location:

Windows Embedded CE Features > Applications and Services > Windows Media Player for Windows Embedded CE

2.8.3 Media Player Fails to Load and Play an Audio File

This problem is typically caused by failing to include the appropriate software codec that is required to handle the audio file format. See the list of recommended audio driver catalog items in Table 2-1 and make sure that support for the desired audio file format is included.

Chapter 3
Backlight Driver

The backlight driver uses the hardware provided by the display module on the device, to control the backlight on the Liquid Crystal Display (LCD) panel. The backlight driver interfaces with the Windows CE Power Manager to provide timed control over the display backlight. A timeout interval controls the length of time that the backlight stays on. The backlight driver is power-manageable, and it meets the requirements of a power-manageable device by implementing the required power management I/O Controls (IOCTLs). The backlight driver uses its own defined timer to set the backlight power states.

3.1 Backlight Driver Summary

Table 3-1 provides a summary of source code location, library dependencies and other BSP information.

Table 3-1. Backlight Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\BACKLIGHT
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\DRIVERS\BACKLIKGHT
Driver DLL	backlight.dll
SDK Library	N\A
Catalog Item	Third Party > BSP > Freescale i.MX233 EVK: ARMV4I > Device Drivers > Backlight
SYSGEN Dependency	SYSGEN_BATTERY=1
BSP Environment Variables	BSP_BACKLIGHT=1

3.2 Supported Functionality

The backlight driver enables the 3-Stack System to provide the following support:

1. Conforms to the Device Manager streams interface
2. Supports 0–10 level adjustment
3. Supports power management mode: full on or full off

3.3 Hardware Operation

This section explains about the hardware operation

3.3.1 i.MX233-EVK Hardware Operation

The hardware consists of a PWM implemented by channel 2 of PWM controller.This PWM is dedicated to drive the backlight of LCD.It can be configured by adjusting the duty cycle in channel 2 of PWM controller.

3.4 Software Operation

The backlight driver is a stream interface driver and is accessed through the file system APIs. To use the backlight driver, a handle to the device must first be created using the **CreateFile** function. Subsequent commands to the device are issued using the **DeviceIoControl** function with IOCTL codes specifying the desired operation.

The control of the backlight operation requires a call to the **DeviceIoControl** function. The following are the possible choices available for the user:

- IOCTL_POWER_CAPABILITIES, register and inform the Power Manager of capabilities
- IOCTL_POWER_QUERY, where the new power state is returned
- IOCTL_POWER_SET, interface to the hardware that controls the backlight through the PDD layer
- IOCTL_POWER_GET, where the current power state is returned

3.4.1 Backlight Driver Registry Settings

This section explains about the backlight driver registry settings.

3.4.1.1 i.MX233-EVK Backlight Driver Registry Setting

The following registry keys are required to properly load backlight driver:

```
[HKEY_CURRENT_USER\ControlPanel\Backlight]
    "BattBacklightLevel"=dword:7F           ; Backlight level settings. 0xFF = Full On
    "ACBacklightLevel"=dword:7F             ; Backlight level settings. 0xFF = Full On
    "BatteryTimeout"=dword:1E               ; 30 seconds
    "ACTimeout"=dword:78                    ; 2 minutes
    "UseExt"=dword:1                        ; Enable timeout when on external power
    "UseBattery"=dword:1                    ; Enable timeout when on battery
    "AdvancedCPL"="AdvBacklight"            ; Enable Advanced Backlight control panel dialog
```

3.4.2 Power Management

The backlight driver consumes power primarily through the operation of the LCD panel backlight. To facilitate the management of this module, the backlight driver implements the IOCTL code IOCTL_POWER_SET.

3.4.2.1 PowerUp

This function is not implemented for the backlight driver.

3.4.2.2 PowerDown

This function is not implemented for the backlight driver.

3.4.2.3 IOCTL_POWER_SET

The backlight driver implements the IOCTL_POWER_SET IOCTL API with support for the D0 (Turn On) and D4 (Set intensity to 0) power states. These states are handled in the following manner:

- D0—Backlight is enabled for LCD panel and the intensity can be adjusted through the PDD layer
- D4—Backlight intensity is set to 0 which is the lowest level of backlight

3.5 Unit Test

The backlight driver is tested by the application test. The following section explains about the hardware and software requirements for unit tests.

3.5.1 Unit Test Hardware

This section explains about the hardware required to run the backlight application test.

3.5.1.1 i.MX233-EVK Unit Test Hardware

Table 3-2 lists the required hardware to run the backlight application test.

Table 3-2. Hardware Requirements

Requirement	Description
SAMSUNG LMS430HF02 WQVGA Panel	Display panel required for display of graphics data

3.5.2 Unit Test Software

Table 3-3 lists the required software to run the backlight application test.

Table 3-3. Software Requirements

Requirement	Description
backlight.dll	The backlight driver to implement the backlight functions
Advbacklight.dll	The file implements adding an Advanced button to the Backlight Control Panel application

3.5.3 Running the Backlight Application Test

Table 3-4 lists the backlight application test.

Table 3-4. Backlight Application Test

Test Case	Entry Criteria/Procedure/Expected Result
Backlight Level	Entry Criteria: N/A Procedure: 1. Go to Setting > Control Panel 2. Double click on the Display icon, then click on the Backlight tab 3. Click on the Advanced… button 4. Modify the backlight level setting for both battery and external power 5. Observe that the backlight level behaves according to the new setting Expected Result: N/A
Backlight Timeout	Entry Criteria: N/A Procedure: 1. Go to Setting > Control Panel 2. Double click on the Display icon, then click on the Backlight tab 3. Modify the backlight timeout setting for both battery and external power, and then click on OK button to apply the changes 4. Observe the time it takes for the backlight to go out, make sure it correspond with the new settings entered in step 3 Expected Result: N/A

3.6 Backlight API Reference

The API for the backlight driver conforms to the stream interface and exposes the standard functions. For more information, see Platform Builder Help at the following location:

Developing a Device Driver > Windows CE Embedded Drivers > Streams Interface Drivers

Chapter 4
Battery Driver

The battery driver module provides information about the battery level to the OS, and decides whether to execute the charging or discharging operation. It will also report battery capability and power supply state to OS periodically by measuring the battery voltage. When charging, current-limit and voltage-limit is maintained to protect the charger and battery.

4.1 Battery Driver Summary

Table 4-1 provides a summary of source code location, library dependencies and other BSP information.

Table 4-1. Battery Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BATTDRVR
Import Library	N/A
Driver DLL	battery.dll
Catalog Item	Third Party > BSP > Freescale i.MX233 EVK:ARMV4I > Device Drivers >Battery
SYSGEN Dependency	SYSGEN_BATTERY
BSP Environment Variables	BSP_NOBATTERY= BSP_BATTERY=1(Only for Real Battery)

4.2 Supported Functionality

The battery driver enables the system to provide the following support:

1. Conforms to the battery driver interface
2. Supports two power management modes, full on and full off
3. Detects power source changes and reports current power source
4. Supports charging of Lion battery
5. Auto stop charging if the die temperature is too high

4.3 Hardware Operation

The battery driver is implemented with the power module of i.MX233. The power module contains on-chip analog to control charging function (including voltage monitor and current limiter). The LRADC channel 7 is used to get the level of voltage in the battery. This level is then used in determining the capacity level of the battery.

4.3.1 Conflicts with Other SoC Peripherals

No conflicts.

4.4 Software Operation

After initialization, the BatteryPDDGetStatus() function is called periodically to get the status of the battery. This function fills the structure SYSTEM_POWER_STATUS_EX2 and returns it to the system. The Power Properties window is updated based on the values in this structure.

4.4.1 Battery Driver Registry Settings

The following registry keys are required to properly load battery driver:

```
; These registry entries load the battery driver.  The IClass value must match
; the BATTERY_DRIVER_CLASS definition in battery.h -- this is how the system
; knows which device is the battery driver.  Note that we are using
; DEVFLAGS_NAKEDENTRIES with this driver.  This tells the device manager
; to instantiate the device with the prefix named in the registry but to look
; for DLL entry points without the prefix.  For example, it will look for Init
; instead of BAT_Init.  This allows the prefix to be changed in the registry (if
; desired) without editing the driver code.
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Battery]
   "Prefix"="BAT"
   "Dll"="battdrvr.dll"
   "Flags"=dword:8                     ; DEVFLAGS_NAKEDENTRIES
   "Order"=dword:3
   "MaxBatteryVoltage"=dword:1068 ; 4200mV
   "BatteryVoltageHighLevel"=dword:E74  ; 3700mV
   "BatteryVoltageLowLevel"=dword:C80   ; 3200mV
   "PollInterval"=dword:1F4; battery polling interval, in milliseonds(0.5 seconds)
   "IClass"="{DD176277-CD34-4980-91EE-67DBEF3D8913}"

[HKEY_LOCAL_MACHINE\System\Events]
    "SYSTEM/BatteryAPIsReady"="Battery Interface APIs"
```

4.4.2 Power Management

There is no additional power management implementation for battery driver.

4.5 Unit Test

The battery driver can be tested, by switching on the system and watching the power properties window. When charging, the charge capacity of the battery can be seen increasing until it is charged to 100%.

NOTE

It is not allowed to plug in or remove out the battery after boot up device.

4.5.1 Unit Test Hardware

The i.MX233-EVK board is required. For real battery mode, switch S12 to the right side and connect a real lion battery to J13 or J21; For fake battery mode, switch S12 to the left side.

4.6 Battery API Reference

The API for the battery driver conforms to the stream interface and exposes the standard functions. For more information, see Platform Builder Help at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Battery Drivers

Chapter 5
Boot from Secure Digital/MultiMedia Card (SD/MMC)

Booting support from SD/MMC includes the following components:

- Boot Image
- Storage for OS binary image (NK)

Boot Image is stored in the SD/MMC card using a special tool, and NK is stored in the FAT partition. The user can select to boot the system from the SD/MMC card, after the booting procedure.

5.1 Boot from SD/MMC Summary

Table 5-1 provides a summary of source code location, library dependencies and other BSP information.

Table 5-1. Boot from SD/MMC Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX233-EVK
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\BOOTLOADER ..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\BOOT\FMD\SDMMC
Driver DLL	N/A
SDK Library	N/A
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	N/A

5.2 Supported Functionality

The Boot support from SD/MMC includes:

1. Supports boot from low or high capacity SD/MMC card
2. Supports storing OS images to SD/MMC flash
3. Supports loading OS image from SD/MMC flash to RAM
4. Supports file system on bootable SD/MMC card

5.3 Hardware Operation

This section explains about the hardware operation of the controller linked with SD/MMC.

5.3.1 Conflicts with Other Peripherals and Catalog Items

On the i.MX233-EVK platform, SSP1 controller is shared between SD/MMC and Ethernet, since both SD and MMC cannot operate at the same time. If the system is booted from SD/MMC card, before using Ethernet download and/or KITL, the card must be removed from the slot in order for Ethernet to function properly. Or, USB-RNDIS can be used for download and KITL to avoid plug or unplug of card.

5.4 Software Operation

On startup while booting from SD/MMC, the Boot ROM is responsible for initializing and bringing the SD/MMC memory to a proper working state. The Boot ROM executes the boot image and boot up the EBOOT, and then passes control to bootloader which in turn brings up the OS.

In the EBOOT, users can select the booting mode to **SDMMC Storage**. Then the EBOOT will read the NK from the FAT partition of the SD/MMC card, and boot up the system.

5.5 Card Flashing Tool

Flashing tool cfimager.exe is used to write the boot image to the SD/MMC. The tool is located in the directory `<%_WINCEROOT%>\SUPPORT\TOOLS\COMMON\CFIMAGER`. Users can follow the instructions in the *readme.txt* file in that folder to write the boot image and boot up the system.

5.5.1 Write Image (EBOOT) to SD Card

Plug SD into Card Reader on PC, and run the following command. The *.sb files to flash are copied to `<%_WINCEROOT%>\SUPPORT\TOOLS\iMX233-EVK\SDIMAGE`. The user can add that path before the filename.

```
cfimager -f eboot.sb -d <card reader drive, no colon>
```

After successful operation, users can copy nk.bin from release directory to <card reader drive>:\.

If users want to boot OS image (NK) without the bootloader (EBOOT), change the command to

```
cfimager -f nk.sb -d <card reader drive, no colon>
```

5.5.2 System Boot

Plug the flashed card into the board, ensure boot switch is set to the defined value and that the appropriate fuses are blown, then power on the board.

Chapter 6
Chip Support Package Driver Development Kit (CSPDDK)

The Chip Support Package Driver Development Kit (CSPDDK) provides an interface to access peripheral features and SOC configuration shared by the system. The CSPDDK executes as a device driver DLL and exports functions for the following SCC components:

- CLOCK
- GPIO
- IOMUX
- DMA(APBH DMA and APBX DMA)
- POWER

6.1 CSPDDK Driver Summary

Table 6-1 provides a summary of source code location, library dependencies and other BSP information.

Table 6-1. CSPDDK Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPDDK
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CSPDDK
Driver DLL	cspddk.dll
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOCSPDDK=

6.2 Supported Functionality

The CSPDDK meets the following requirements:

1. Supports an interface that allows synchronized inter-process access to the following set of shared SoC resources:
   — IOMUX (DDK_IOMUX)
   — GPIO (DDK_GPIO)
   — DMA (DDK_APBHDMA and DDK_APBXDMA)

- CLK (DDK_CLK)
- POWER(DDK_POWER)

2. Exposes exported functions that can be invoked without incurring a system call (for example, not a stream driver)

6.3 Hardware Operation

See the hardware specification document for detailed operation and programming information.

6.3.1 Conflicts with Other Peripherals and Catalog Items

This section explains about the CSPDDK conflicts with other peripherals and catalog items.

6.3.1.1 Conflicts with SoC Peripherals

The following section explains about the CSPDDK conflicts with SoC peripherals.

6.3.1.1.1 iMX233 Peripheral Conflicts

See the i.MX233 hardware specification document for possible conflicts.

6.3.1.2 Conflicts with Hardware Peripherals

No conflicts.

6.4 Software Operation

This section explains about the CSPDDK software operation.

6.4.1 Communicating with the CSPDDK

Similar to the CEDDK DLL, the CSPDDK DLL does not require any special initialization. All of the initialization required by the CSPDDK is performed when the DLL is loaded into the respective process space. Drivers that want to utilize the CSPDDK simply need to link to the CSPDDK export library and invoke the exported functions.

6.4.2 Compile-Time Configuration Options

No options.

6.4.3 Registry Settings

There are no registry settings that need to be modified to use the CSPDDK driver. Since most drivers need to use CSPDDK functionality, the CSPDDK should be one of the first DLLs loaded by Device Manager.

6.4.4 Power Management

The CSPDDK exposes interfaces that allow drivers to self-manage power consumption by controlling clocking and pin configuration. The CSPDDK executes as a shared DLL and does not implement the Power Manager driver IOCTLs or the PowerUp or PowerDown stream interface. However, the CSPDDK functions are invoked by other drivers during power state transits.

6.5 Unit Test

Due to the heavy use of the CSPDDK routines by other drivers on the system, currently there is no additional test case.

6.5.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

The DDK_CLK interface allows device drivers to configure and query system clock settings.

6.5.1.1 DDK_CLK Enumerations

Table 6-2 lists all the programming elements in the DDK_CLK enumerations.

Table 6-2. DDK_CLK Enumerations

Programming Element	Description
DDK_CLOCK_SIGNAL	Clock signal name for querying/setting clock configuration
DDK_CLOCK_GATE_INDEX	Index for referencing the corresponding clock gating control bits within the CCM
DDK_CLOCK_GATE_MODE	Clock gating modes supported by CCM clock gating registers
DDK_CLOCK_BAUD_SOURCE	Input source for baud clock generation
DDK_DVFC_SETPOINT	Frequency/voltage setpoints supported by the DVFC driver

6.5.1.2 DDK_CLK Functions

The following are the functions that are used to set DDK_CLK.

6.5.1.2.1 DDKClockSetGatingMode

This function sets the clock gating mode of the peripheral.

```
BOOL DDKClockSetGatingMode(
        DDK_CLOCK_GATE_INDEX index,
        DDK_CLOCK_GATE_MODE mode)
```

Parameters

index [in] Index for referencing the peripheral clock gating control bits

mode [in] Requested clock gating mode for the peripheral

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.2 DDKClockGetGatingMode

This function retrieves the clock gating mode of the peripheral.

```
BOOL DDKClockGetGatingMode(
        DDK_CLOCK_GATE_INDEX index,
        DDK_CLOCK_GATE_MODE *pMode)
```

Parameters

index	[in] Index for referencing the peripheral clock gating control bits
pMode	[out] Current clock gating mode for the peripheral
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.3 DDKClockGetFreq

This function retrieves the clock frequency in Hz for the specified clock signal.

```
BOOL DDKClockGetFreq(
        DDK_CLOCK_SIGNAL sig,
        UINT32 *freq)
```

Parameters

sig	[in] Clock signal
freq	[out] Current frequency in Hz
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.4 DDKClockSetFreq

This function sets the clock frequency in Hz for the specified clock signal.

```
BOOL DDKClockSetFreq(
        DDK_CLOCK_SIGNAL sig,
        UINT32 freq)
```

Parameters

sig	[in] Clock signal.
freq	[in] Requested frequency in Hz.
Return Values	Returns TRUE if successful, otherwise returns FALSE.

6.5.1.2.5 DDKClockConfigBaud

This function configures the input source clock and dividers for the specified CCM peripheral baud clock output.

```
BOOL DDKClockConfigBaud(
        DDK_CLOCK_SIGNAL sig,
        DDK_CLOCK_BAUD_SOURCE src,
        UINT32 preDiv,
```

```
        UINT32 postDiv)
```

Parameters

sig	[in] Clock signal to configure
src	[in] Selects the input clock source
preDiv	[in] Specifies the value programmed into the baud clock predivider
postDiv	[in] Specifies the value programmed into the baud clock postdivider
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.6 DDKClockSetpointRequest

This function requests the DVFC driver to transition to a setpoint that meets or exceeds the voltage and clocking requirements of the setpoint being requested. This function optionally blocks until the setpoint request has been granted.

```
BOOL DDKClockSetpointRequest(
        DDK_DVFC_SETPOINT setpoint,
        BOOL bBlock)
```

Parameters

setpoint	[in] Specifies the setpoint to be requested
bBlock	[in] Set TRUE to block until the setpoint has been granted; set FALSE to return immediately after the request has been submitted
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.7 DDKClockSetpointRelease

This function releases a setpoint previously requested using DDKClockSetpointRequest.

```
BOOL DDKClockSetpointRelease(
        DDK_DVFC_SETPOINT setpoint)
```

Parameters

setpoint	[in] Specifies the setpoint to be released
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.1.2.8 DDKClockGetSharedConfig

This function obtains a reference to the global shared clock configuration data structure. This is intended to be used by the DVFC driver.

```
PDDK_CLK_CONFIG DDKClockGetSharedConfig(VOID)
```

Parameters	None
Return Values	Returns a pointer to the clock configuration data structure.

6.5.1.2.9 DDKClockLock

This function requests a lock of the global shared clock configuration data structure.

```
VOID DDKClockLock(VOID)
```

Parameters None

Return Values None

6.5.1.2.10 DDKClockUnLock

This function releases a lock of the global shared clock configuration data structure.

```
VOID DDKClockUnLock(VOID)
```

Parameters None

Return Values None

6.5.1.3 DDK_CLK Examples

The following are the example code for the DDK_CLK.

Example 6-1 shows the sample code for CSPDDK clock gating.

Example 6-1. CSPDDK Clock Gating

```
#include “csp.h”    // Includes CSPDDK definitions

// Enable I2C1 peripheral clock
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_I2C1, DDK_CLOCK_GATE_MODE_ENABLED_ALL);

// Disable I2C1 peripheral clock
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_I2C1, DDK_CLOCK_GATE_MODE_DISABLED);
```

Example 6-2 shows the sample code for CSPDDK clock rate query.

Example 6-2. CSPDDK Clock Rate Query

```
#include “csp.h”    // Includes CSPDDK definitions

UINT32 freq;

// Query the current bus clock
DDKClockGetFreq(DDK_CLOCK_SIGNAL_AHB, &freq);
```

6.5.2 CSPDDK DLL GPIO (DDK_GPIO) Reference

The DDK_GPIO interface allows device drivers to utilize the GPIO ports. Each GPIO port has a single interrupt request line that is shared for all port pins. In addition, configuration, status, and data registers are shared. The DDK_GPIO provides safe access to the shared GPIO resources.

6.5.2.1 DDK_GPIO Enumerations

Table 6-3 lists all the programming elements in the DDK_GPIO enumerations.

Table 6-3. DDK_GPIO Enumerations

Programming Element	Description
DDK_GPIO_BANK	Specifies the GPIO module instance
DDK_GPIO_CFG	Specifies the configuration of the GPIO pins

6.5.2.2 DDK_GPIO Functions

The following section explains about the DDK_GPIO functions.

6.5.2.2.1 DDKGpioConfig

This function configures the gpio_pin as input/output, sets the drive strength, voltage, and as interrupt selection (if applicable).

```
BOOL DDKGpioConfig(DDK_IOMUX_PIN gpio_pin,
                   DDK_GPIO_CFG gpio_cfg,
                   DDK_IOMUX_PAD_DRIVE drive,
                   DDK_IOMUX_PAD_VOLTAGE voltage,
                   BOOL bPull_Enable)
```

Parameters

gpio_pin [in] functional pin name

gpio_cfg [in] structure to configure the pin as input/output, interrupt selection.

drive [in] set the gpio_pin drivestrength.

voltage [in] set the gpio_pin voltage.

bPull_Enable [in] enable/disable the pullup for the gpio_pin.

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.2.2.2 DDKGpioEnableDataPin

This function sets the value in the register bit to drive on the gpio_pin.

```
BOOL DDKGpioEnableDataPin(DDK_IOMUX_PIN pin,UINT32 data)
```

Parameters

pin [in] functional pin name

data [in] data to be written to the pin.

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.2.2.3 DDKGpioWriteDataPin

This function sets the value in the register bit to drive on the gpio_pin.

```
BOOL DDKGpioWriteDataPin(DDK_IOMUX_PIN pin,UINT32 data)
```

Parameters

pin	[in] functional pin name
data	[in] data to be written to the pin.
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.2.2.4 DDKGpioReadDataPin

This function reads the GPIO port data from the specified pin.

```
BOOL DDKGpioReadDataPin(DDK_IOMUX_PIN pin, UINT32 *pData)
```

Parameters

pin	[in] GPIO pin [0-31].
pData	[out] points to buffer for data read. Data will be shifted to LSB.
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.2.2.5 DDKGpioReadIntr

This function reads register for the interrupt status.

```
BOOL DDKGpioReadIntr(DDK_IOMUX_PIN pin, UINT32 *pData)
```

Parameters

pin	[in] functional pin name
pData	[out] pointer to the data read.
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.3 CSPDDK DLL IOMUX (DDK_IOMUX) Reference

The DDK_IOMUX interface allows device drivers to configure signal multiplexing and pad configuration. This control resides inside the IOMUX registers and is shared for the entire system. The DDK_IOMUX support allows drivers to dynamically update and query their signal multiplexing and pad configuration.

6.5.3.1 DDK_IOMUX Enumerations

Table 6-4 lists all the programming elements in the DDK_IOMUX enumerations.

Table 6-4. DDK_IOMUX Enumerations

Programming Element	Description
DDK_IOMUX_PIN	Specifies the functional pin name used to configure the IOMUX.
DDK_IOMUX_PIN_MUXMODE	Specifies the mux mode for a signal

Table 6-4. DDK_IOMUX Enumerations (continued)

Programming Element	Description
DDK_IOMUX_PAD_DRIVE	Specifies the drive strength for a pad; if no DRIVE bit for a PAD, the DDK_IOMUX_PAD_DRIVE_NULL should be set.
DDK_IOMUX_PAD_PULL	Specifies the pull-up/pull-down/keeper configuration for a pad
DDK_IOMUX_PAD_VOLTAGE	Specifies the driver voltage for a pad, either 1.8 V or 3.3 V

6.5.3.2 DDK_IOMUX Functions

This sections explains about the DDK_IOMUX functions.

6.5.3.2.1 DDKIomuxSetPinMux

This function sets the IOMUX mux for the specified IOMUX pin.

```
BOOL DDKIomuxSetPinMux(DDK_IOMUX_PIN pin, DDK_IOMUX_PIN_MUXMODE muxmode)
```

Parameters

pin [in] functional pin name used to configure IOMUX HW_PINCTRL_MUXSEL

muxmode [in] MUX_MODE configuration.

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.3.2.2 DDKIomuxGetPinMux

This function gets the IOMUX mux configuration for the specified IOMUX pin.

```
BOOL DDKIomuxGetPinMux(DDK_IOMUX_PIN pin, DDK_IOMUX_PIN_MUXMODE *pMuxmode)
```

Parameters

pin [in] functional pin name used to select the IOMUX output or input path that will be returned.

pMuxmode [out] MUX_MODE configuration.

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.3.2.3 DDKIomuxSetPadConfig

This function sets the IOMUX pad configuration for the specified IOMUX pad.

```
BOOL DDKIomuxSetPadConfig(DDK_IOMUX_PIN pin,
                          DDK_IOMUX_PAD_DRIVE drive,
                          DDK_IOMUX_PAD_PULL pull,
                          DDK_IOMUX_PAD_VOLTAGE voltage)
```

Parameters

pad [in] functional pad name used to select the pad that will be configured.

drive	[in] drive strength configuration.
pull	[in] pull-up, pull-down, or keeper configuration.
voltage	[in] drive voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

6.5.3.2.4 DDKIomuxEnablePullup

This function enables the IOMUX pad configuration for the specified IOMUX pad.

```
BOOL DDKIomuxEnablePullup(DDK_IOMUX_PIN pin, BOOL bEnable)
```

Parameters

pin	[in] functional pin name used to select the IOMUX output or input path that will be returned.
bEnable	[in] enable or disable pullup
Return Values	Returns TRUE if successful, otherwise returns FALSE.

6.5.3.2.5 DDKIomuxGetPadConfig

This function gets the IOMUX pad configuration for the specified IOMUX pad.

```
BOOL DDKIomuxGetPadConfig(DDK_IOMUX_PIN pin,
                          DDK_IOMUX_PAD_DRIVE   *pDrive,
                          DDK_IOMUX_PAD_PULL    *pPull,
                          DDK_IOMUX_PAD_VOLTAGE *pVoltage)
```

Parameters

pin	[in] functional pin name used to select the IOMUX output or input path that will be returned
pDrive	[out] drive strength configuration
pPull	[out] pull-up, pull-down, or keeper configuration
pVoltage	[out] drive voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

6.5.4 CSPDDK DLL DMA (DDK_DMA) Reference

The DDK_DMA interface allows device drivers to allocate, configure, and control shared DMA resources.

6.5.4.1 DDK_DMA Functions

This section explains about the DDK_DMA functions.

6.5.4.1.1 DDKApbhStartDma

This function loads the NEXTCOMMAND address and increments the semaphore to start the DMA operation for first command.

```
BOOL DDKApbhStartDma(UINT8 Channel,PVOID memAddrPA, UINT8 semaphore)
```

Parameters

Channel	[in] channel number
memAddrPA	[in] pointer of memory's physical address
semaphore	[in] DMA semaphore
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.2 DDKApbhStopDma

This function stops the DMA channel.

```
BOOL DDKApbhStopDma(UINT8 Channel)
```

Parameters

Channel	[in] channel number
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.3 DDKApbhDmaInitChan

This function initializes the requested DMA channel.

```
BOOL DDKApbhDmaInitChan(UINT8 Channel,BOOL bEnableIrq)
```

Parameters

Channel	[in] channel number
bEnableIrq	[in] enable or disable the irq
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.4 DDKApbhDmaChanCLKGATE

This function clears the interrupt for respective channel.

```
BOOL DDKApbhDmaChanCLKGATE(UINT8 Channel,BOOL bClockGate)
```

Parameters

Channel	[in] channel number
bClockGate	[in] gate or un-gate the channel
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.5 DDKApbhDmaClearCommandCmpltIrq

This function clears the interrupt for respective channel.

```
BOOL DDKApbhDmaClearCommandCmpltIrq(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.6 DDKApbhDmaEnableCommandCmpltIrq

This function enables the interrupt for respective channel.

```
BOOL DDKApbhDmaEnableCommandCmpltIrq(UINT8 Channel,BOOL bEnable)
```

Parameters

Channel [in] channel number

bEnableIrq [in] enable or disable the interrupt for respective channel

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.7 DDKApbhDmaResetChan

This function resets the AHB to APBH bridge channel based on the argument channel.

```
BOOL DDKApbhDmaResetChan(UINT8 Channel,BOOL bReset)
```

Parameters

Channel [in] channel number

bReset [in] reset or un-reset the channel

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.8 DDKApbhDmaFreezeChan

This function freezes the AHB to APBH bridge channel based on the argument channel.

```
BOOL DDKApbhDmaFreezeChan(UINT8 Channel,BOOL bFreeze)
```

Parameters

Channel [in] channel number

bFreeze [in] freeze or un-freeze the channel

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.9 DDKApbhDmaGetPhore

This function gets the phore of respective channel.

```
UINT32 DDKApbhDmaGetPhore(UINT32 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns the virtual channel index if successful, otherwise returns NULL**.**

6.5.4.1.10 DDKApbxStartDma

This function loads the NEXTCOMMAND address and increments the semaphore to start the DMA operation for first command.

```
BOOL DDKApbxStartDma(UINT8 Channel,PVOID memAddrPA, UINT8 semaphore)
```

Parameters

Channel [in] channel number

memAddrPA [in] pointer of memory's physical address

semaphore [in] DMA semaphore

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.11 DDKApbxGetNextCMDAR

This function gets the NEXTCOMMAND address.

```
UINT32 DDKApbxGetNextCMDAR(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.12 DDKApbxStopDma

This function stops the DMA channel.

```
BOOL DDKApbxStopDma(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.13 DDKApbxDmaInitChan

This function initializes the requested DMA channel.

```
BOOL DDKApbxDmaInitChan(UINT8 Channel,BOOL bEnableIrq)
```

Parameters

Channel [in] channel number

bEnableIrq [in] enable/disable the irq

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.14 DDKApbxDmaGetActiveIrq

This function gets the active irq status of DMA channel.

```
BOOL DDKApbxDmaGetActiveIrq(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.15 DDKApbxDmaClearCommandCmpltIrq

This function clears the interrupt for respective channel.

```
BOOL DDKApbxDmaClearCommandCmpltIrq(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.16 DDKApbxDmaClearErrorIrq

This function clears the error interrupt for respective channel.

```
BOOL DDKApbxDmaClearErrorIrq(UINT8 Channel)
```

Parameters

Channel [in] channel number

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.17 DDKApbxDmaEnableCommandCmpltIrq

This function enables the interrupt for respective channel.

```
BOOL DDKApbxDmaEnableCommandCmpltIrq(UINT8 Channel,BOOL bEnable)
```

Parameters

Channel [in] channel number

bEnableIrq [in] enable/disable the interrupt for respective channel

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.18 DDKApbxDmaEnableErrorIrq

This function enables the interrupt for respective channel.

```
BOOL DDKApbxDmaEnableErrorIrq(UINT8 Channel,BOOL bEnable)
```

Parameters

Channel	[in] channel number
bEnableIrq	[in] enable or disable the interrupt for respective channel
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.19 DDKApbxDmaResetChan

This function resets the AHB to APBX bridge channel based on the argument channel.

```
BOOL DDKApbxDmaResetChan(UINT8 Channel,BOOL bReset)
```

Parameters

Channel	[in] channel number
bReset	[in] reset or un-reset the channel
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.4.1.20 DDKApbxDmaFreezeChan

This function freezes the AHB to APBX bridge channel based on the argument channel.

```
BOOL DDKApbxDmaFreezeChan(UINT8 Channel,BOOL bFreeze)
```

Parameters

Channel	[in] channel number
bFreeze	[in] freeze or un-freeze the channel
Return Values	Returns TRUE if successful, otherwise returns FALSE

6.5.5 CSPDDK POWER (DDK_POWER) Reference

The DDK_POWER interface allows device drivers to configure and control DC-DC converter, linear regulators, PSWITCH pin functions, battery monitor and charger, and silicon speed sensor.

6.5.5.1 DDK_POWER Functions

The following are the functions of DDK_POWER.

6.5.5.1.1 DDKPowerInit

This function initializes the power functionality for the supporting modules.

```
BOOL DDKPowerInit(POWER_INITVALUES* pInitValues)
```

Parameters

pInitValues [in] pointer to the structure POWER_INITVALUES

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.5.1.2 DDKPowerEnableDcdc

This function enables ENABLE_DCDC.

```
VOID DDKPowerEnableDcdc(BOOL bEnable)
```

Parameters

bEnable [in] TRUE for enable, FALSE for disable

Return Values Returns TRUE if successful, otherwise returns FALSE

6.5.5.1.3 DDKPowerExecuteBatteryTo5VoltsHandoff

This function hands off power source from battery to 5Volts supply.

```
VOID DDKPowerExecuteBatteryTo5VoltsHandoff(VOID)
```

Parameters None

Return Values None

6.5.5.1.4 DDKPowerExecute5VoltsToBatteryHandoff

This function hands off power source from 5Volts supply to battery.

```
void DDKPowerExecute5VoltsToBatteryHandoff(void)
```

Parameters None

Return Values None

6.5.5.1.5 DDKPowerEnable5VoltsToBatteryHandoff

This function enables the handoff from 5Volts supply to battery.

```
VOID DDKPowerEnable5VoltsToBatteryHandoff(VOID)
```

Parameters None

Return Values None

6.5.5.1.6 DDKPowerEnableBatteryTo5VoltsHandoff

This function enables the handoff from battery to 5Volts.

```
void DDKPowerEnableBatteryTo5VoltsHandoff(void)
```

Parameters None

Return Values None

6.5.5.1.7 DDKPowerGet5vPresentFlag

This function checks if the 5V supply is present.

```
BOOL DDKPowerGet5vPresentFlag(VOID)
```

Parameters None

Return Values If the 5V supply is present returns TRUE, otherwise returns FALSE

6.5.5.1.8 DDKPowerInitPowerSupplies

This function initializes the power supplies.

```
UINT32 DDKPowerInitPowerSupplies(VOID)
```

Parameters None

Return Values Returns 0 if successful otherwise returns error value.

6.5.5.1.9 DDKPowerGetDirectBoot

This function checks if direct boot.

```
BOOL DDKPowerGetDirectBoot(void)
```

Parameters None

Return Values If the direct boot returns TRUE, otherwise returns FALSE

6.5.5.1.10 DDKPowerInitBatteryMonitor

This function initializes the battery monitor for battery module.

```
UINT32 DDKPowerInitBatteryMonitor(LRADC_DELAYTRIGGER eTrigger, UINT32 SamplingInterval)
```

Parameters

eTrigger [in] Specifies the Lradc trigger used

SamplingInterval [in] Specifies the sampling interval for the Battery value to be sampled

Return Values Returns 0 if successful else returns error values

6.5.5.1.11 DDKPowerGetBatteryMode

This function returns the current Battery Mode.

```
POWER_BATTERYMODE DDKPowerGetBatteryMode(VOID)
```

Parameters None

Return Values Returns the battery mode.

6.5.5.1.12 DDKPowerSetCharger

This function is used to set the charger.

```
VOID DDKPowerSetCharger(DWORD current)
```

Parameters

current [in] The current value of charger

Return Values None

6.5.5.1.13 DDKPowerStopCharger

This function is used to stop the charger.

```
VOID DDKPowerStopCharger()
```

Parameters None

Return Values None

6.5.5.1.14 DDKPowerGetBatteryVoltage

This function returns the current Battery voltage.

```
UINT16 DDKPowerGetBatteryVoltage(VOID)
```

Parameters None

Return Values Returns the battery voltage

6.5.5.1.15 DDKPowerGetBatteryChargingStatus

This function returns the Battery charging status.

```
BOOL DDKPowerGetBatteryChargingStatus(VOID)
```

Parameters None

Return Values Returns battery charging status

6.5.5.1.16 DDKPowerClear5VIrq

This function is used to clear the 5V irq

```
VOID DDKPowerClear5VIrq()
```

Parameters None

Return Values None

6.5.5.1.17 DDKPowerGetLimit

This function is used to get the limit status.

```
BOOL DDKPowerGetLimit()
```

Parameters None

Return Values If the power limit returns TRUE, otherwise returns FALSE

6.5.5.1.18 DDKPowerGetUSBPhy

This function is used to get the USB PHY information.

```
BOOL DDKPowerGetUSBPhy()
```

Parameters None

Return Values If the USB PHY is plug in returns TRUE, otherwise returns FALSE

6.5.5.1.19 DDKPowerSetVdddValue

This function sets Vddd value.

```
void DDKPowerSetVdddValue(UINT16 u16Vddd_mV)
```

Parameters

u16Vddd_mV [in] Convert voltage value (mv) to register setting

Return Values None

6.5.5.1.20 DDKPowerGetVdddValue

This function gets Vddd value.

```
UINT32 DDKPowerGetVdddValue(VOID)
```

Parameters None

Return Values Returns Vddd current voltage value

6.5.5.1.21 DDKDumpPowerRegisters

This function dumps the power registers used for debugging.

```
VOID DDKDumpPowerRegisters(VOID)
```

Parameters None

Return Values None

6.5.5.1.22 DDKPowerSetLimit

This function is used to set the limit information.

```
VOID DDKPowerSetLimit(void)
```

Parameters None

Return Values None

6.5.5.1.23 DDKPowerGetPSwitchIrq

This function is used to get the PSwitch irq.

```
BOOL DDKPowerGetPSwitchIrq()
```

Parameters None

Return Values If the PSwitch irq in returns TRUE, otherwise returns FALSE

6.5.5.1.24 DDKPowerClearPSwitchIrq

This function is used to clear the PSwitch irq.

```
VOID DDKPowerClearPSwitchIrq()
```

Parameters None

Return Values None

6.5.5.1.25 DDKPowerGetPSwitchStatus

This function is used to get the PSwitch status.

```
DWORD DDKPowerGetPSwitchStatus()
```

Parameters None

Return Values Returns the PSwitch status

Chapter 7
Configurable Serial Peripheral Interface (CSPI) Driver

The CSPI module provides master functionality of a standard CSPI bus.

7.1 CSPI Driver Summary

Table 7-1 provides a summary of source code location, library dependencies and other BSP information.

Table 7-1. CSPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPI
Platform Driver Path	..\PLATFORM\<*Target Platform*>\DRIVERS\CSPI
Import Library	cspisdk.lib
Driver DLL	cspi.dll
Catalog Item	Third Party > BSP > Freescale <*TGTPLAT*> > Device Drivers > CSPI Bus 1
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SSP1_CSPI = 1

7.2 Supported Functionality

The CSPI driver supports the following features:

1. Supports the CSPI master mode of operation
2. Supports CSPI configurable bus feature
3. Supports configurable access method of polling method
4. Supports stream interface
5. Supports two power management modes: full on and full off

7.2.1 Conflicts with Other Peripherals and Catalog Items

This section explains about the conflicts that the CSPI driver have with other peripherals and catalog items.

7.2.1.1 Conflicts with SoC Peripherals

On i.MX233-EVK platform, SSP module is shared by CSPI driver and SD/MMC driver, so these two drives can not be enabled at the same time.

7.2.2 Conflicts with EVK Peripherals

No conflicts.

7.3 Software Operation

This section explains about the software operation for the CSPI module.

7.3.1 Registry Settings

The following registry keys are required to properly load the CSPI module.

```
;; CSPI Bus Driver
;;
IF BSP_SSP1_CSPI
IF BSP_NOSSP1_SDHC
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI1]
   "Prefix"="SPI"
   "Dll"="cspi.dll"
   "Index"=dword:1
ENDIF ;BSP_NOSSP1_SDHC
ENDIF ;BSP_SSP1_CSPI
```

7.3.2 Communicating with the CSPI

The CSPI is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the CSPI, a handle to the device must first be created using the **CreateFile** function. Subsequent commands to the device are issued using the **DeviceIoControl** function with IOCTL codes specifying the desired operation. If preferred, the **DeviceIoControl** function calls can be replaced with macros that hide the **DeviceIoControl** call details. The following are the basic steps:

7.3.3 Creating a Handle to the CSPI

Call the **CreateFile** function to open a connection to the CSPI device. A CSPI port must be specified in this call. The format is **SPIX:**, with X being the number indicating the CSPI port. This number should not exceed the number of CSPI instances on the platform. If an CSPI port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the CSPI:

1. Insert a colon after the CSPI port for the first parameter, *lpFileName*

   For example, specify SPI1: as the CSPI port
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an CSPI port are supported by the driver.

3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

Example 7-1 is a sample code to open a CSPI port.

Example 7-1. Code to open CSPT port

```
// Open the serial port.
hSPI = CreateFile (L"SPI1:",                                  // name of device
                   GENERIC_READ | GENERIC_WRITE,              // access (read-write) mode
                   FILE_SHARE_READ | FILE_SHARE_WRITE,        // sharing mode
                   NULL,                                      // security attributes (ignored)
                   OPEN_EXISTING,                             // creation disposition
                   FILE_FLAG_RANDOM_ACCESS,                   // flags/attributes
                   NULL);                                     // template file (ignored)
```

7.3.4 Data Transfer Operations

The CSPI driver provides one command, SPIExchange, that facilitates performing both reads and writes through the CSPI bus. The basic unit of data transfer in the CSPI driver is the CSPI_XCH_PKT, which contains a buffer for reading and writing data, and a CSPI_BUSCONFIG datum that specifies the desired bus configuration and XCH method which is used during the SPI transmission. The following steps explain the process of performing write and read operations through the CSPI bus.

Before these actions can be taken, a handle to the CSPI port must already be opened. Each of these steps requires a call to the **DeviceIoControl** function. As parameters, the CSPI port handle, appropriate IOCTL code, and other input and output parameters are required.

To perform an CSPI transfer:

1. Create a CSPI_XCH_PKT object and initialize the fields of the packet as follows:
   a) Initialize a CSPI_BUSCONFIG datum to specify the bus parameters as SSPCTRL0, SSPCMD, SSPARG, BITCOUNT, bREAD, and specify the method parameters for use/not use DMA, use/not use POLLING, send/not send command.
   b) Set the *pBuf* field to the user buffer which sends and receives data.
   c) Set the *xchCnt* field, for the 1-8 bit XCH, the *xchCnt = bytes*, for the 9-16 bit XCH, the *xchCnt = words*, for the 17-32 bit XCH, the *xchCnt = dwords*.
2. Set the *hDevice* parameter to the previously acquired CSPI port handle.
3. Set the *dwIoControlCode* to the SPI_IOCTL_EXCHANGE IOCTL code.
4. Set the *lpInBuffer* to point to the CSPI_XCH_PKT object created in step 1. Set *nInBufferSize* to the size of that packet object.
5. Set *lpOutBuffer*, *lpBytesReturned*, and *lpOverlapped* to NULL. Set *nOutBufferSize* to 0.

Example 7-2 demonstrates how to perform a XCH transfer.

Example 7-2. Code for XCH transfer

```
CSPI_BUSCONFIG_T buscnfg =
{
    0x9000004,                      //configuration for SSP control register 0
```

```
    0,                                  //command 0
    0,                                  //command 1
    FALSE,                              //no DMA
    TRUE,                               //polling mode
    32,                                 //32 bits data
    FALSE,                              //write data
    0,                                  //no command will be sent
};

DWORD Data[11];

CSPI_XCH_PKT_T xchPkt =
{
    &buscnfg,
    Data,
    1,                                  // XCH to target SPI device 1 times
    NULL,
    0
};
                                        // Transfer data via CSPI
DeviceIoControl(hCSPI,                  // file handle to the driver
     CSPI_IOCTL_EXCHANGE,               // I/O control code
                (PBYTE) &xchPkt,        // in buffer
                sizeof(xchPkt),         // in buffer size
                NULL,                   // out buffer
                0,                      // out buffer size
                NULL,                   // number of bytes returned
                NULL);                  // ignored (=NULL)
```

As a substitute for the **DeviceIoControl** call above, a SDK wrap function may be used to simplify the code. The following is the sample code:

```
CSPIExchange(hCSPI, &xchPkt);
```

7.3.5 Closing the Handle to the CSPI

Call the **CloseHandle** function to close a handle to the CSPI after an application finishes using it. **CloseHandle** has one parameter, which is the handle returned by the **CreateFile** function call that opened the CSPI port.

7.3.6 Power Management

The primary method for limiting power consumption in the CSPI module is to gate off the input clock to the module when the input CSPI clock is not needed. This is accomplished through the **DDKClockSetGatingMode** function call. In all of the BSP use cases, the CSPI controller acts as a master device. As a result, the CSPI clock can be turned off, whenever the module is not processing CSPI packets.

As described in the **Data Transfer Operations** section, the CSPI driver turns on the CSPI clocks and enables the CSPI module before processing an CSPI XCH, and then disables and turns off clocks to the CSPI module after the XCH has been done. This limits the time during which the CSPI module is consuming power to the time during which the CSPI is actively performing data transfers.

7.3.6.1 PowerUp

This function is not implemented for the CSPI driver. Power to the CSPI module is managed as CSPI transfer operations are processed. There are no additional power management steps needed for the CSPI.

7.3.6.2 PowerDown

This function is not implemented for the CSPI driver.

7.3.6.3 IOCTL_POWER_SET

This function is implemented for the CSPI driver. When D4 power mode is set, the driver switches its operating mode to polling mode that does not produce interrupt events to BSP system. When leaving the D4 power mode, the driver recovers its origin operating mode.

7.4 Unit Test

The CSPI driver does not use the CETK for unit testing, but uses the test program described in the following section for unit tests.

7.4.1 Building the Unit Tests

To build the CSPI tests, build an OS image for the desired configuration using these steps:

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the SPI Test directory: `\WINCE600\SUPPORT\TEST\SPITEST`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the EXE to the flat release directory.
4. Input **build -c** to build SPI test.

After the build completes, the SPIAPP.EXE file is located in the $(_FLATRELEASEDIR) directory.

To run the application within VS2005 use the following steps:

1. Go to the Target menu option and select the Run Programs menu option. This gives a list of applications that can be run on the OS.
2. Select SPIAPP.EXE from this list.
3. Click on Run to run this application.

7.5 CSPI Driver API Reference

This section explains about the CSPI driver API reference.

7.5.1 CSPI Driver IOCTLs

This section consists of descriptions for the CSPI I/O control codes (IOCTLs). These IOCTLs are used in calls to **DeviceIoControl** to issue commands to the CSPI device. Only relevant parameters for the IOCTL have a description provided.

7.5.1.1 CSPI_IOCTL_EXCHANGE

This **DeviceIoControl** request performs the transfer of data to a target device. An CSPI_XCH_PKT object is required, which contains CSPI bus configuration parameters and data buffers. All of the required information should be stored in the CSPI_XCH_PKT passed in the *lpInBuffer* field.

Parameters

lpInBuffer	Pointer to an CSPI_XCH_PKT structure containing a pointer to bus configuration parameters and data buffers
nInBufferSize	Size in bytes of the CSPI_XCH_PKT

7.5.2 CSPI Driver SDK Wrapper

This section explains about the CSPI driver SDK wrapper.

7.5.2.1 CSPIOpenHandle

This function retrieves the CSPI device handle.

```
HANDLE CSPIOpenHandle(
        LPCWSTR lpDevName
);
```

Parameters

lpDevName	The CSPI device name for retrieving handle from CreateFile()
Return Values	Returns Handle for CSPI driver; returns INVALID_HANDLE_VALUE if failure

7.5.2.2 CSPICloseHandle

This function closes a handle of the CSPI stream driver.

```
BOOL CSPICloseHandle(
        HANDLE hDev
);
```

Parameters

hDev	The CSPI device handle retrieved from CreateFile()
Return Values	Returns TRUE or FALSE. If the result is TRUE, the operation is successful

7.5.2.3 CSPIExchange

This function performs XCH operations.

```
BOOL CSPITransfer(
        HANDLE hDev,
        PCSPI_XCH_PKT_T pCspiXchPkt
```

```
    );
```

Parameters

hDev	The CSPI device handle retrieved from CreateFile()
pCspiXchPkt	[in] Pointer to XCH packet with bus configuration parameters
Return Values	Returns TRUE or FALSE. If the result is TRUE, the operation is successful

7.5.3 CSPI Driver Structures

This section explains about the CSPI driver structures.

7.5.3.1 CSPI_BUSCONFIG_T

This structure contains the bus configuration information needed during CSPI performs XCH.

```
// CSPI bus configuration
typedef struct
{
    SSP_CTRL0 SspCtrl0;
    SSP_CMD0  SspCmd;
    SSP_CMD1  SspArg;
    BOOL      usedma;
    BOOL      usepolling;
    UINT8     bitcount;
    BOOL      bRead;
    BOOL      bCmd;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Table 7-2 shows the CSPI_BUSCONFIG_T structure members.

Table 7-2. CSPI_BUSCONFIG_T Structure Members

Member	Description
SspCtrl0	Configuration for SSP control register 0
SspCmd	Command 0 for SSP
SspArg	Command 1for SSP
usedma	If TRUE, uses DMA mode, not support DMA now
usepolling	If TRUE, uses polling mode, only support polling mode now
bitcount	Define bits used in a single XCH, range 1-32.
bRead	If TRUE, read data from CSPI
bCmd	If TRUE, send command 0 and command 1 to CSPI controller
usedma	If TRUE, uses DMA mode
usepolling	If TRUE, uses polling mode

7.5.3.2 CSPI_XCH_PKT_T

This structure contains an XCH buffer parameters to be used in data exchange to CSPI device.

```
// CSPI exchange packet
```

```
typedef struct
{
    PCSPI_BUSCONFIG_T pBusCnfg;
    LPVOID pBuf;
    UINT32 xchCnt;
    LPWSTR xchEvent;
    UINT32 xchEventLength;
} CSPI_XCH_PKT_T, *PCSPI_XCH_PKT_T;
```

Table 7-3 shows the CSPI_XCH_PKT_T structure members.

Table 7-3. CSPI_XCH_PKT_T Structure Members

Member	Description
pBusCnfg	A pointer to CSPI bus configuration object
pBuf	A pointer to data buffer
xchCnt	Amount of XCH operation to SPI device
xchEvent	Asynchronous access using the internal exchange queue
xchEventLength	Event name length including tailing Zero

Chapter 8
Display Driver for LCDIF and PXP

The Windows Embedded CE 6.0 BSP display driver is based on the Microsoft DirectDraw Graphics Primitive Engine (DDGPE) classes, and supports the Microsoft DirectDraw interface. This driver combines the functionality of a standard LCD display with DirectDraw support. The display driver interfaces with the LCD Interface (LCDIF) and Pixel Pipeline (PXP).

The display driver supports the following display type:

- SAMSUNG LMS430HF02 WQVGA Panel

8.1 Display Driver Summary

Table 8-1 identifies the source code location, library dependencies and other BSP information for the display driver.

Table 8-1. Display Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\LCDIF ..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\PXP
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DISPLAY
Driver DLL	ddi_stmp37xx_chilin.dll
Import Library	ddgpe.lib, gpe.lib
Catalog Items	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > SAMSUNG LMS430HF02(WQVGA)
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_NODISPLAY=

8.2 Supported Functionality

The display driver enables the 3-Stack board to provide the following software and hardware support:

1. RGB565 user interface
2. DirectDraw Hardware Abstraction Layer (DDHAL)
3. One overlay surface

4. Video overlays containing image data in any of the following FOURCC pixel formats: RGB565, YV12
5. Hardware-accelerated color space conversion in video overlays
6. Hardware-accelerated image resizing in video overlays
7. Overlay surface color key feature
8. Alpha blending with an overlay surface
9. Two power management modes: full on and full off (resume and suspend)
10. Screen rotation
11. Cropping of an overlay surface
12. Supports SAMSUNG LMS430HF02 WQVGA Panel

NOTE

The following limitations apply to the display driver overlay support.

13. RGB image resize is not supported
14. Cropping is not supported while performing alpha blending operation
15. The width and height of the overlay surface must conform to an 8-pixel alignment restriction
16. The minimum width (or height if screen is rotated) of an overlay surface is 8 pixels
17. The minimum height (or width if screen is rotated) of an overlay surface is 8 pixels
18. When using the cropping feature, the x and y coordinate position must conform to 8-pixel alignment restriction

8.3 Hardware Operation

For operation and programming information, see the chapter on the Pixel Pipeline and LCD Interface in the Reference Manual.

8.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

8.4 Software Operation

This section explains about the software operation of the display driver.

8.4.1 Software Driver Components

Figure 8-1 shows the block diagram explaining the relationship between the software components in the display driver architecture.

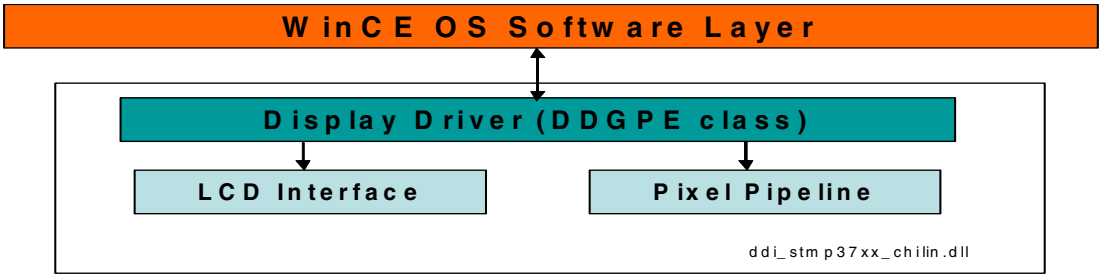

Figure 8-1. Software Driver Components Block Diagram

A list of the main elements of the display driver architecture is as follows:

- **Display Driver**—The high level DDGPE-based display driver. Contains implementations for DirectDraw APIs.
- **LCD Interface—**Set of functions provide access to LCDIF module for display control setting.
- **Pixel Pipeline Driver**—A stream interface driver that performs the following processing tasks: color space conversion, resizing, rotation, and combining.

8.4.1.1 Display Driver

The display driver is the top level interface between the display driver and the Windows CE OS or a calling application. This top level software component is composed of the `Stmp37xxDDGPE` class, which is derived from the public DDGPE class and inherits the underlying GPE driver functionality. Graphics Device Interface (GDI) and DirectDraw APIs are implemented at this level.

8.4.1.2 LCD Interface

The LCDIF software component consists of a collection of functions that provide access to the LCDIF module registers. These functions are called from the display driver to implement the display control. The major tasks that this component performs include the following:

- Setting bus master and DMA operation modes for LCD.
- Configuring LCD data bus depending on the packet size.
- Programming timing and parameters to support a wide variety of displays.

8.4.1.3 Pixel Pipeline

The PXP driver provides a general resource that is capable of performing a set of processing tasks on a surface:

- Resizing
- Combining of video and graphics data

- Rotation
- Vertical and horizontal flipping
- Color Space Conversion (CSC)
- Cropping

The PXP driver is the primary means for performing resizing, rotation, CSC, cropping and combining on an overlay surface.

8.4.2 Communicating with the Display

Communication with the display driver is accomplished through Microsoft-defined APIs. A framework for accessing the display driver is provided through the Graphics Device Interface (GDI) and DirectDraw.

8.4.2.1 Using the GDI

The GDI provides basic controls for the display of text and graphics. For more information, see the Help in the following location:

Windows Embedded CE Features > Shell, GWES and User Interface > Graphics, Windowing and Events (GWES) > GWES Application Development > Graphics Device Interface (GDI)

8.4.2.2 Using DirectDraw

The DirectDraw API provides support for hardware-accelerated 2-D graphics, offering fast access to display hardware while retaining compatibility with the GDI. For information about the DirectDraw API, see the DirectDraw Help or the MSDN documentation library in the following location:

Windows Embedded CE Features > Graphics > DirectDraw

The following DirectDraw features are supported in the display driver by the PXP hardware:

- Page flipping with one backbuffer.
- Overlay surfaces using RGB or YV12 pixel format.
- Overlaying using a color key for the overlay surface for RGB colors.
- Stretching of overlay surfaces.

The PXP hardware module is used within the display driver to accelerate the following operations:

- Color space conversion of YUV overlay data to RGB. This conversion may be required in order to combine the overlay data with RGB graphics plane data before being displayed.
- Resizing of the overlay surface.
- Rotation of the overlay surface (used when the screen orientation is rotated).

8.4.2.3 Using Display Driver Escape Codes

In some cases, applications might need to communicate directly with a display driver. To make this possible, an escape code mechanism is provided as a part of the display driver.

For a detailed description of standard display driver escape codes, see the CE Help in the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Display Drivers > Display Driver Development Concepts > Display Driver Escape Codes

8.4.3 Configuring the Display

The display configuration is based on the **GUID** registry key, which is described in Section 8.4.3.2, "Display Registry Settings." The **GUID** registry key indicates the display panel that is being used. The only one supported display panel is the SAMSUNG LMS430HF02 WQVGA Panel.

8.4.3.1 Rotation Support

The DirectDraw display driver supports screen rotation.

NOTE

Due to lack of support for the co-existence of GDI screen rotation and DirectDraw, a DirectDraw display driver with rotation support may yield more failures in the GDI or DIRECTDRAW CETK test suite. It is recommended to run these CETK tests under 0 rotation degree. See the Windows CE Help, stating that GDI screen rotation cannot be used with DirectDraw.

8.4.3.2 Display Registry Settings

A set of registry keys is included in the OS image, depending on the display panel catalog item included in the OS design.

8.4.3.2.1 i.MX233 Registry Settings

If the SAMSUNG LMS430HF02 WQVGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\System\GDI\Drivers]
"GUID"="{1ED47D96-6842-4c20-8705-BF05AD0DFC33}"
```

8.4.4 Power Management

The display driver implements the power management I/O Control (IOCTL) codes, such as IOCTL_POWER_CAPABILITIES, IOCTL_POWER_QUERY, IOCTL_POWER_GET and IOCTL_POWER_SET.

8.4.4.1 PowerUp

This function is implemented in the PXP driver. It enables clock gating with the PXP module and resets the PXP module to its default state. If an PXP operation is previously terminated by calling the PowerDown() API, the PowerUp() function will restore the PXP module registers and restart the PXP operation.

8.4.4.2 PowerDown

This function is implemented in the PXP driver. If the PowerDown() function is called while PXP operations are going on, it will store current PXP module registers setting temporarily. Then it disables clock gating with the PXP module and holds the PXP module in its reset (lower power) state.

8.4.4.3 IOCTL_POWER_SET

The display driver implements the IOCTL_POWER_SET IOCTL API with support for the D0 (Full On) and D4 (Off) power states. These states are handled in the following manner:

- D0—The LCDIF module is enabled. The display panel is enabled. Clock gating is enabled for clocks to the LCDIF.
- D4—The LCDIF module is disabled. The display panel is disabled. Clock gating is disabled for clocks to the LCDIF.

8.5 Unit Test

The display driver is subject to two test suites provided with the Windows CE Test Kit (CETK): the GDI Test and the DirectDraw Test. Additionally, the video playback is verified using the Windows Media Player application.

The GDI Test is designed to test a graphics device interface. This test verifies that basic shapes, including rectangles, triangles, circles, and ellipses, are drawn correctly. The test also examines the color palette of the display, verifies that the display is correctly divided into multiple regions, and tests whether a device context can be properly created, stored, retrieved, and destroyed.

The DirectDraw Test analyzes basic DirectDraw functionality including block image transfers (blits), scaling, color keying, color filling, flipping, and overlaying.

Windows Media Player may be used to play back WMV video files and visually verify correct operation of video overlays, accelerated color space conversion, and accelerated image resizing.

8.5.1 Unit Test Hardware

The SAMSUNG LMS430HF02 WQVGA panel is needed to run the GDI and DirectDraw tests. The panel displays the graphics data.

8.5.2 Unit Test Software

This section explains about the software required for different tests.

8.5.2.1 GDI Tests

Table 8-2 lists the software required to run the GDI tests.

Table 8-2. GDI Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
Gdiapi.dll	Main test.dll file
Ddi_test.dll	Graphics Primitive Engine (GPE)–based display driver that the GDI API uses to verify the success of each test case. If Ddi_test.dll is unavailable, run the test with manual verification

8.5.2.2 DirectDraw Tests

Table 8-3 lists the software required to run the DirectDraw tests.

Table 8-3. DirectDraw Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
DDrawTK.dll	Test.dll file

8.5.2.3 Windows Media Player Tests

Table 8-4 lists the software required to perform WMV playback with Windows Media Player.

Table 8-4. Windows Media Player Software Requirements

Requirement	Description
Ceplayer.exe	Windows Media Player sample application
*.wmv sample video files	Sample windows media files

8.5.3 Building the Unit Tests

The GDI and DirectDraw tests come pre-built as part of the CETK. Ensure that the latest CETK suite is installed. No steps are required to build these tests. For information about the tests, see the Help at the following location:

Windows Embedded CE Test Kit > Running the CETK

For Windows Media Player testing, there are no build steps required. The Windows Media Player catalog item must be added to the OS image to ensure that `ceplayer.exe` is included in the image. Additionally, sample WMV files must be included in the image to demonstrate playback.

8.5.4 Running the Unit Tests

This section explains how to run different types of tests.

8.5.4.1 Running the GDI Tests

The command for running the GDI tests is:

```
tux –o –d gdiapi.dll -c "/NoResize"
```

For information about the GDI tests and command line options, see the Platform Builder Help topic:

Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Display Tests > Graphics Device Interface Test

8.5.4.2 Running the DirectDraw Tests

The command for running the DirectDraw tests is:

```
tux –o –d ddrawtk
```

NOTE

The display driver fails the following DirectDraw CETK test cases: 1240, 1340. The failure occurs because the hardware can not support RGB image resize, and the failing tests perform RGB pixel format overlay surfaces resize that violate this restriction.

8.5.4.3 Running the Windows Media Player tests

The command for starting playback of a WMV test video clip in Windows Media Player is:

```
ceplayer [wmv test file]
```

For example, `ceplayer motocross_208x160_30fps.wmv`

If audio support is not included in the current BSP, the message **Audio hardware is missing or disabled** appears when the WMV file is being loaded. Click **OK** to continue to WMV playback.

To confirm the correct operation of this test, observe the application and verify whether the video clip have a clear image, normal coloring, and correct image sizing.

8.6 Display Driver API Reference

For information about the display driver APIs, see CE Help. No additional custom API information is required for the features currently supported in the display driver.

For reference information on basic display driver functions, methods, and structures, see the CE Help at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Display Drivers > Display Driver Reference

For reference information on DirectDraw functions, callbacks, and structures, see the CE Help at the following location:

Windows Embedded CE Features > Graphics > DirectDraw

Chapter 9
Dynamic Voltage and Frequency Control (DVFC) Driver

The BSP includes the DVFC driver that provides combined support for DVFS (Dynamic Voltage Frequency Scaling). The DVFC driver plays an important role in the reduction of active power consumption by dynamically adjusting the voltage and frequency settings of the system. The DVFC driver responds to DVFC hardware logic or load tracking software that is monitoring CPU loading and process/temperature performance of the silicon.

9.1 DVFC Driver Summary

Table 9-1 provides a summary of source code location, library dependencies, and other BSP information.

Table 9-1. DVFC Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2_
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\DVFC
SOC Specific Path	
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC
Driver DLL	dvfc.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > DVFC driver support using the MC13892
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_DVFC = 1

9.2 Supported Functionality

The DVFC driver enables the hardware platform to provide the following software and hardware support:

1. Executes as a device driver and provides synchronized support of the DVFS power management feature
2. Exposes stream interface for initialization and power management
3. Supports D0 and D4 driver power states and support control of frequency or voltage setpoint based on Power Manager device power states
4. Supports peripheral setpoint requests initiated by CSPDDK clock management code

9.2.1 i.MX233 Supported Functionality

1. Supports DVFS for CPU and AHB
2. Supports reactive CPU load tracking to control setpoint based on system performance requirements. Current release uses software load tracking algorithm
3. Provides voltage control using PMU

9.3 Hardware Operation

This section describes about the DVFC hardware operation.

9.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

9.3.2 i.MX233 EVK Configuration

The DVFC driver is dependent DDK Power interface for dynamic voltage control through PMU.

9.4 Software Operation

This section describes about the registry settings.

9.4.1 i.MX233 Registry Settings

The following registry keys are required to properly load the i.MX233 DVFC module.

```
;-------------------------------------------------------------------------------
; DVFC Driver
;
IF BSP_DVFC
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC]
   "Prefix" = "DVF"
   "Index" = dword:1
   "Dll"="dvfc.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
ENDIF ;BSP_DVFC
```

9.4.2 Loading and Initialization

The DVFC driver is automatically loaded to kernel space by the Device Manager as a stream driver. As part of the loading procedure of stream drivers, the device manager invokes the corresponding stream initialization function exported by the DVFC driver. The initialization sequence includes a call to platform-specific code `(BSPDvfcInit)` to allow the OEM to configure and tune the DVFC driver operation.

9.4.3 Operation

The DVFC driver is the central point in the BSP for controlling voltage and frequency scaling. The DVFC communicates with the PMIC and CCM to coordinate the DVFS. The DVFC driver responds to setpoint

requests from DDK_CLK (by driver calling **DDKClockSetGatingMode**) and Power Manager (by **IOCTL_POWER_SET**). A shared global data structure (**DDK_CLK_CONFIG**) is used to keep track of reference counts for each setpoint. The DVFC relies on synchronization with the DDK_CLK component to determine when it is safe to transition to a new setpoint. DVFC integration with the Power Manager allows drivers and applications direct control of the setpoint by using the **SetDevicePower** API.

9.4.3.1 i.MX233 Voltage or Frequency Setpoints

The i.MX233 DVFC driver supports mutually exclusive voltage and frequency setpoints for the CPU power domains. Table 9-2 provides the voltage/frequency characteristics for these setpoints.

Table 9-2. i.MX233 DVFC Setpoints

Setpoint Name	CPU/AHB Frequency [MHz]	VDDD Voltage
DDK_DVFC_SETPOINT_HIGH	454.74/151.58	1.55 V
DDK_DVFC_SETPOINT_MEDIUM	266.82/130.91	1.275 V
DDK_DVFC_SETPOINT_LOW	160/80	1.275 V

The setpoint attributes are controlled by the definitions in the platform-specific DVFS header file (found in `\PLATFORM\<Target Platform>\SRC\INC\dvfs.h`). The DVFC driver uses these definitions to populate a global setpoint array (g_SetPointConfig) that is referenced during setpoint transitions.The EMI clock frequency is fixed to 96MHz, not changed with the setpoint change.

9.4.3.2 i.MX233 Setpoint Mapping

The DVFC driver advertises support for IOCTL_POWER requests from Power Manager. A IOCTL_POWER_SET request is mapped to a setpoint by the DVFC driver. This mapping allows applications to use the Power Manager APIs to request changes in the DVFC setpoint. The mapping of device power states (D0-D4) to DVFC setpoints is located in **DvfcMapDevPwrStateToSetpoint** (found in `\PLATFORM\<Target Platform>\SRC\DRIVERS\DVFC\dvfc.c`). To change the setpoint mapping for a specific device power state (D0-D4), modify the code in **DvfcMapDevPwrStateToSetpoint**.

9.4.4 DDK Interface

The DVFC driver allows other drivers or applications to control some aspects of the DVFS operation. Due to the tight coupling with the system clock configuration, this interface is exposed within CSPDDK clocking support. See the CSPDDK documentation for the following functions:

- DDKClockSetpointRequest, Section 6.5.1.2.6, “DDKClockSetpointRequest.”
- DDKClockSetpointRelease, Section 6.5.1.2.7, “DDKClockSetpointRelease.”

9.4.5 Power Management

The DVFC is an integral part of the power management supported by the BSP. However, since the DVFC runs as a driver on the system, it also supports the Power Manager device driver interface. This allows the DVFC driver to be notified of when the system is suspending or resuming and configure the processor performance accordingly.

9.4.5.1 PowerUp

This stream interface function is not implemented for the DVFC driver.

9.4.5.2 PowerDown

This stream interface function is not implemented for the DVFC driver.

9.4.5.3 IOCTL_POWER_CAPABILITIES

The DVFC driver advertises that D0–D4 device power states are supported.

9.4.5.4 IOCTL_POWER_SET

The DVFC driver supports requests to enter D0–D4 device power state.

9.4.5.5 IOCTL_POWER_GET

The DVFC driver reports the current device power state (D0, D1, D2 or D4).

9.5 Unit Test

This section explains about the unit testing.

9.5.1 i.MX233 Unit Testing

A stress test application for the DVFC driver is provided for unit testing. This stress test uses the Power Manager interface (**SetDevicePower**) to randomly request setpoints for the CPU and peripheral DVFS domains. Follow these steps to run this unit test.

1. Open <*Target Platform*>-Mobility workspace and add the DVFC driver catalog item. Build OS image.

   NOTE

   Note that modifications to the default workspace may cause additional drivers to be included and may prevent the system from transitioning through all possible DVFS setpoints.

2. Build DVFC stress test located in \SUPPORT\TEST\APP\PWRMGMT. The resulting application pwrmgmt.exe is generated in the flat release directory.
3. Boot the OS image and launch application code such as media player that can perform continuous playback. WMA audio playback is a good use case since audio playback can be performed across all supported setpoints.
4. Launch the stress test application. From the CE shell, the stress test can be launched with the following command line:

   ```
   s \release\pwrmgmt.exe
   ```

Board modifications are required to observe voltage setpoints and are not covered in this document. Debug messages to indicate setpoint transitions can be enabled using the DVFC_VERBOSE macro found in `\PLATFORM\<Target Platform>\SRC\DRIVERS\DVFC\dvfc.c`

Chapter 10
Keypad Driver

The keypad driver converts input from the sensor into keyboard events that the driver enters into the Graphics, Windowing, and Events Subsystem (GWES).

10.1 Keypad Driver Summary

Table 10-1 provides a summary of source code location, library dependencies and other BSP information.

Table 10-1. Keypad Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\KEYBD
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\KEYPAD
Driver DLL	keypad.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX233 EVK: ARMV4I > Device Drivers > KEYPAD> KEYPAD
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOKEYPAD=

10.2 Supported Functionality

The Keypad driver enables the hardware platform to provide the following software and hardware support:

1. Conforms to the Microsoft Layout Manager Interface
2. Two power management modes, full on and full off

10.3 Hardware Operation

See the chapter on the Low-Resolution ADC (LRADC) in the hardware specification document for detailed operation and programming information.

10.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

10.3.2 Keypad

The keypad driver interfaces with the Windows CE Keyboard Driver Architecture to provide key input support.

10.3.2.1 i.MX233 EVK Keypad Mapping

The 8-key keypad is located on the personality board and the mapping is shown in Table 10-2.

Table 10-2. 8-key Keypad Mapping

LABEL	Key
KEY1	ESC
RIGHT	RIGHT
KEY2	WIN
LEFT	LEFT
UP	UP
DOWN	DWON
KEY3	MENU
SELECT	ENTER

10.4 Software Operation

The i.MX233 Keypad driver does not follows the Microsoft-recommended architecture for keyboard drivers. Use a standard stream driver for keypad scan.

10.4.1 Keypad Scan Codes and Virtual Keys

Each key on the keypad has a unique scan code, which is added to a buffer whenever that key is pressed or released. These scan codes, which are hardware specific, are converted to intermediate PS/2 keyboard scan code values and then converted into virtual keys, which are hardware independent numbers that identify the key. If a key is pressed from the keyboard, the generated scan code is directly converted into virtual keys.

10.4.1.1 i.MX233 EVK Scan Code Mapping

Table 10-3 shows the scan code mapping.

Table 10-3. I.MX233 EVK Scan Code Mapping Table

Key	Keypad Voltage	Virtual Key
KEY1	0x0	VK_ESCAPE
RIGHT	0x156	VK_RIGHT

Table 10-3. I.MX233 EVK Scan Code Mapping Table (continued)

Key	Keypad Voltage	Virtual Key
KEY2	0x2AA	VK_RWIN
LEFT	0x3F4	VK_LEFT
UP	0x6B0	VK_UP
DOWN	0x960	VK_DOWN
KEY3	0x80C	VK_MENU
SELECT	0xC00	VK_ENTER

10.4.2 Power Management

The following are the power management functions used by the keypad driver.

10.4.2.1 BSPKppPowerOn

This function is used to power up the keypad. This function configures the necessary settings in the registers to bring up the keypad.

10.4.2.2 BSPKppPowerOff

This function powers down the keypad.

10.4.2.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the keypad driver.

10.4.2.4 IOCTL_POWER_SET

This function is not implemented for the keypad driver.

10.4.2.5 IOCTL_POWER_GET

This function is not implemented for the keypad driver.

10.4.3 Keypad Registry Settings

The following registry keys are required to properly load the keypad device layout and input language.

```
IF BSP_KEYPAD
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\KEYPADS]
   "Prefix"="KPD"
   "Dll"="keypad.dll"
   "Index"=dword:1
   "Order"=dword:6

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\KeyPadS]
   "RepeatLantency"=dword:3e8
```

```
   "RepeatRate"=dword:a
   "ScanPeriod"=dword:14
   "Debounce"=dword:2
   "ValidKeys"=dword:b
   "LRADC_KeyPAD"=dword:0
   "LRADC_Reference"=dword:6
   "LRADC_SCHEDULER"=dword:0
   "Hysteresis"=dword:80
   "ReleaseVolatge"=dword:00000e4a

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\KeyPadS\Key2]
   "KeyName"="HotKey0"
   "AppName"="explorer.exe"
   "Parameter"=""
   "Voltage"=dword:00000550
   "Flag"=dword:00000001
   "VirtualKey"=dword:0000ffff

ENDIF ; BSP_KEYPAD
```

10.5 Unit Test

As keypad has only 8 keys, it’s not a full-key keypad, it can’t pass the Keyboard Test included in the Windows CE Test Kit (CETK). A specific manual test to verify the 8-key functionality is described in following sections.

10.5.1 Unit Test Hardware

- i.MX233 EVK board

10.5.2 Unit Test Software

The manual keypad test requires Microsoft WordPad which can be built into the image.

10.5.3 Building the Unit Tests

No additional steps are required to build the keypad tests.

10.5.4 Running the Unit Tests

The procedure of keyboard tests is as follows:

1. Input Enter to run the Internet Explorer application
2. Input Menu
3. Input Up Down Left and Right
4. Input Windows Key
5. Open the help document by click the question mark on Internet Explorer application
6. Input the ESC to quit from help document
7. Input Return to quit the Explorer application

NOTE

Prior this test, make sure the WordPad items is included in the project (SYSGEN_PWORD).

Chapter 11
Inter-Integrated Circuit (I^2C) Driver

The Inter-Integrated Circuit (I^2C) module provides the functionality of a standard I^2C master. The I^2C module is designed to be compatible with the standard Phillips I^2C bus protocol.

11.1 I^2C Driver Summary

Table 11-1 provides a summary of source code location, library dependencies and other BSP information.

Table 11-1. I^2C Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\iMX233_FSL_V2\I2C
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\I2C
Platform Driver Path	..\PLATFORM*Target Platform*>\SRC\DRIVERS\I2C
Import Library	N/A
Driver DLL	i2csdk.dll i2c.dll
Catalog Item	Third Party > BSP > Freescale <TGTPLAT> > Device Drivers > I2CBus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_I2CBUS=1

11.2 Supported Functionality

The I^2C driver supports the following features:

1. I^2C communication protocol
2. I^2C master mode of operation
3. Stream interface

11.3 Hardware Operation

The i.MX233 I2C block has its own dedicated DMA channel in the APBX controller. DMA is used exclusively to transfer data to and from the bus as PIO mode is not fully supported in hardware.

11.3.1 Conflicts with Other Peripherals and Catalog Items

The following section explains about the conflicts that the I^2C driver have with other peripherals and catalog items:

11.3.1.1 Conflicts with SoC Peripherals

No conflicts.

11.3.1.2 Conflicts with Board Peripherals

No conflicts.

11.4 Software Operation

Only master mode is implemented in the driver; slave functions are stubbed out. As mentioned above, PIO mode is not fully supported in hardware so only DMA mode is implemented in the driver. The driver allocates its own DMA buffers for the data transfer. The calling application is expected to setup the data buffer with the slave address (7-bit or 10-bit) as part of the data to be sent in the format required by the slave device.

The I^2C APIs should be used to perform any operation on or using the I^2C module. Any array of packets to be transferred to or from the I^2C bus finish to completion without preemption by another request to transfer data.

11.4.1 Registry Settings

This section explains about the registry settings for the I^2C driver.

11.4.1.1 i.MX233 Registry Settings

The following is the registry key to load the I^2C.

```
IF BSP_I2CBUS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:1
   "Order"=dword:3
ENDIF ; BSP_I2CBUS
```

11.4.2 Communicating with the I^2C

The I^2C is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the I^2C, a handle to the device must first be created using the **CreateFile** function. Subsequent commands to the device are issued using the **DeviceIoControl** function with IOCTL codes specifying the desired operation. The following are the basic steps. The I^2C driver is provided to hide all the IOCTL calls from the calling application.

11.4.3 Creating a Handle

Call the **CreateFile** function to open a connection to the I2C device. An I2C port must be specified in this call. The port to be opened is **I2C1:** since there is only 1 instance of the controller on i.MX233. If an I2C port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the I2C:

1. Insert a colon after the I2C port for the first parameter, *lpFileName*. For example, specify `I2C1:`.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an I2C port are supported by the driver.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

Example 11-1shows how to open an I2C port.

Example 11-1. Code to Open I2C Port

```
// Open the I2C port.
hI2C = CreateFile (“I2C1:”,                                  // name of device
                   GENERIC_READ | GENERIC_WRITE,             // access (read-write) mode
                   FILE_SHARE_READ | FILE_SHARE_WRITE,       // sharing mode
                   NULL,                                     // security attributes (ignored)
                   OPEN_EXISTING,                            // creation disposition
                   FILE_FLAG_RANDOM_ACCESS,                  // flags/attributes
                   NULL);                                    // template file (ignored)
```

Before writing to or reading from an I2C port, configure the port. When an application opens an I2C port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

11.4.4 Configuring the I2C

Configuring the I2C port for communications involves two main operations:

- Setting the master mode
- Setting the I2C clock rate

Before these actions can be taken, a handle to the I2C port must already be opened. Each of these steps requires a call to the **DeviceIoControl** function. As parameters, the I2C port handle, appropriate IOCTL code, and other input and output parameters are required. Use the helper APIs to correctly configure the port.

Example 11-2 shows the code to configure an I2C port:

Example 11-2. Code to Configure I2C Port

```
HANDLE hI2C = I2COpenHandle(_T("I2C1:"));

if (hI2C == INVALID_HANDLE_VALUE)
{
        ERRORMSG(1, (L"Unable to open handle to I2C block\r\n"));
```

```
        retVal = -1;
        goto exit;
}

if (!I2CSetMasterMode(hI2C))
{
        ERRORMSG(1, (L"Unable to set master mode\r\n"));
        retVal = -1;
        goto exit;
}

if (!I2CSetClockRate(hI2C, EEPROM_CLOCK_RATE))
{
        ERRORMSG(1, (L"Unable to set clock rate\r\n"));
        retVal = -1;
        goto exit;
}
```

11.4.5 Data Transfer Operations

The I2C driver provides one command, transfer, that facilitates performing both reads and writes through the I2C. The basic unit of data transfer in the I2C driver is the I2C_PACKET, which contains a buffer for reading or writing data and a flag that specifies whether the desired operation is a read or a write. An array of these packets makes up an I2C_TRANSFER_BLOCK object, which is needed to perform a Transfer operation. The steps below detail the process of performing write and read operations through the I2C.

Before these actions can be taken, a handle to the I2C port must already be opened, and it should already be configured in the correct mode with the correct frequency.

To perform an I2C transfer:

1. Create an array of I2C_PACKET objects and initialize the fields of each packet as follows:
   a) Set the *byRW* field to I2C_RW_WRITE to specify that the I2C operation is a write, or I2C_RW_READ to specify that the I2C operation is a read.
   b) If *byRW* is set to I2C_RW_WRITE, create a buffer of bytes and fill it with the data to write to the slave device. Set the *pbyBuf* field to point to this buffer. If *byRW* is set to I2C_RW_READ, create a buffer of bytes to hold the data which is read from the slave device. The data buffer should include the slave address as required. For reads, an initial write packet with the slave address may need to be inserted, resulting in more than 1 packet to execute the read operation.
   c) Set the *wLen* field to the size, in bytes, of the read or write buffer. This indicates the number of bytes to write or read.
   d) Set the *lpiResult* field to point to an integer that holds the return value from the write operation.
2. Call the I2CTransfer SDK API to start the I2C transfer.
3. After calling the I2CTransfer function, check the *lpiResult* field if the function returned FALSE, to narrow down the type of error that occurred.

Example 11-3 demonstrates how to perform a transfer that contains one write of 3 bytes, which consists of the EEPROM address+direction byte, followed by 2 bytes of internal subaddress of the EEPROM where subsequent data writes will be stored.

Example 11-3. Code to perform I^2C transfer

```
BYTE pSendBytes[3];
I2C_TRANSFER_BLOCK I2CXferBlock;
I2C_PACKET I2CPacket;
INT iResult;

// addr of EEPROM + high byte + low byte of addr within the memory
pSendBytes[0] = EEPROM_I2C_ADDR | 0;
pSendBytes[1] = EEPROM_SUB_ADDR_HI_BYTE;
pSendBytes[2] = EEPROM_SUB_ADDR_LO_BYTE;

I2CXferBlock.pI2CPackets = I2CPacket;
I2CXferBlock.iNumPackets = 1;

I2CPacket.wLen = 3;
I2CPacket.byRW = I2C_RW_WRITE;
I2CPacket.pbyBuf = pSendBytes;
I2CPacket.lpiResult = &iResult;
if (!I2CTransfer(hI2C,&I2CXferBlock))
    {
        ERRORMSG(1, (_T("Write transfer to EEPROM failed!\r\n")));
        retVal = -1;
        goto exit;
    }
```

11.4.6 Closing the Handle

Call the **CloseHandle** function to close the handle to the I^2C after the transfer task is complete. **CloseHandle** has one parameter, which is the handle returned by the **CreateFile** function call that opened the I^2C port.

11.5 Unit Test

The following section explains about the hardware and software requirements for unit tests.

11.5.1 Unit Test Hardware

This section explains about the hardware test.

11.5.1.1 I2C EEPROM Test

24LC512 or 24LC256 EEPROM module which can be read/write by I^2C.

11.5.2 Unit Test Software

This section explains about the software test.

11.5.2.1 I2C EEPROM Test

Please read the following file for information on running the EEPROM test:
`WINCE600\SUPPORT\APP\EEPROM_24LCxxx_I2C_Test\readme.txt`

11.5.3 Building the Unit Tests

The following section explains how to build the unit tests.

11.5.3.1 I2C EEPROM Tests

To build the I2C MCU tests, build an OS image for the desired configuration using these steps:

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the I2C EEPROM tests directory: `\WINCE600\SUPPORT\APP\EEPROM_24LCxxx_I2C_Test`
3. Input **build -c** to build the I2C EEPROM test.

After the build completes, the eepromi2c.exe file is located in the $(_FLATRELEASEDIR) directory.

11.5.4 Running the Unit Tests

The following section explains how to run the unit tests.

11.5.4.1 I2C EEPROM Tests

Run the application using the following command line: `eepromi2c 256` for 24LC256 or `eepromi2c 512` for the 24LC512 model. If no argument is specified then 24LC256 is assumed.

If the test passes, the message **EEPROM Test completed successfully!** is displayed on the console.

11.6 Hardware Limitations

The following is the hardware limitations:

PIO mode is not supported by the hardware; DMA mode is always used. Slave mode is not implemented.

11.7 I^2C Driver API Reference

This section explains about the reference to I^2C driver API.

11.7.1 I^2C Driver IOCTLS

This section contains descriptions of the I^2C I/O control codes (IOCTLs). These IOCTLs are used in calls to **DeviceIoControl** to issue commands to the I^2C device. Only relevant parameters for the IOCTL have a description provided.

11.7.1.1 I2C_IOCTL_GET_CLOCK_RATE

This **DeviceIoControl** request retrieves the clock rate

Parameters

lpOutBuffer	Pointer to the clock rate.
nOutBufferSize	Size in bytes of the clock rate

11.7.1.2 I2C_IOCTL_GET_SELF_ADDR

This **DeviceIoControl** request retrieves the address of the I^2C device. This macro is only meaningful if it is currently in Slave mode.

Parameters

lpOutBuffer	Pointer to the current I^2C device address, valid range is [0x00–0x7F]
nOutBufferSize	Size in bytes of the I^2C device address

11.7.1.3 I2C_IOCTL_IS_MASTER

This **DeviceIoControl** request determines whether the I^2C is currently in Master mode.

Parameters

lpOutBuffer	Pointer to a BYTE that contains the return value from the Master mode inquiry: TRUE if currently in Master mode; FALSE if currently in Slave mode
nOutBufferSize	Size in bytes of the return value, should be one byte

11.7.1.4 I2C_IOCTL_IS_SLAVE

This **DeviceIoControl** request determines whether the I^2C is currently in Slave mode.

Parameters

lpOutBuffer	Pointer to a BYTE that contains the return value from the Slave mode inquiry: TRUE if currently in Slave mode; FALSE if currently in Master mode
nOutBufferSize	Size in bytes of the return value, should be one byte

11.7.1.5 I2C_IOCTL_RESET

This **DeviceIoControl** request performs a hardware reset. The I^2C driver maintains all of the current information of the device, including all of the initialized addresses.

11.7.1.6 I2C_IOCTL_SET_CLOCK_RATE

This **DeviceIoControl** request initializes the I^2C device with the given clock rate. Note that only two frequencies are supported: 100 khz and 400 khz. If a value less than 100 khz is passed in, then the clock rate is setup for 100 khz, and if a value greater than 100 khz is passed in, then 400 khz is setup.

Parameters

lpInBuffer Pointer to the clock rate.

nInBufferSize Size in bytes of the clock rate

11.7.1.7 I2C_IOCTL_SET_MASTER_MODE

This DeviceIoControl request sets the I^2C device to Master mode.

11.7.1.8 I2C_IOCTL_SET_SELF_ADDR

This DeviceIoControl request initializes the I^2C device with the given address.

Parameters

lpInBuffer Pointer to the expected I^2C device address, valid range is [0x00–0x7F]

nInBufferSize Size in bytes of the I^2C device address

Remarks The device expects to respond when any master on the I^2C bus wishes to proceed with any transfer. This IOCTL has no effect if the I^2C device is in Master mode.

11.7.1.9 I2C_IOCTL_SET_SLAVE_MODE

This **DeviceIoControl** request sets the I^2C device to Slave mode.

NOTE

IOCTL is stubbed out because slave mode is not supported.

11.7.1.10 I2C_IOCTL_TRANSFER

This **DeviceIoControl** request performs the transfer (read or write) of one or more packets of data to a target device. An I2C_TRANSFER_BLOCK object is expected, which contains an array of I2C_PACKET objects to be executed sequentially. All of the required information should be stored in the I2C_TRANSFER_BLOCK passed in the lpInBuffer field.

Parameters

lpInBuffer Pointer to an I2C_TRANSFER_BLOCK structure containing a pointer to an array of I2C_PACKET objects specifying all of the information required to perform the requested Read and Write operations

nInBufferSize Size in bytes of the I2C_TRANSFER_BLOCK

11.7.1.11 I2C_IOCTL_ENABLE_SLAVE

This **DeviceIoControl** request starts the I^2C device to work in slave mode.

NOTE

IOCTL is stubbed out because slave mode is not supported.

11.7.1.12 I2C_IOCTL_DISABLE_SLAVE

This **DeviceIoControl** request stops the I^2C device to work in slave mode.

11.7.2 I^2C Driver SDK Encapsulation

This section explains about the functions that are involved in I^2C driver SDK encapsulation.

11.7.2.1 I2COpenHandle

This function retrieves the I^2C device handle.

```
HANDLE I2COpenHandle(
        LPCWSTR lpDevName);
```

Parameters

lpDevName The I^2C device name for retrieving handle from CreateFile()

Return Values Returns the handle for I^2C driver, returns INVALID_HANDLE_VALUE if failure

11.7.2.2 I2CCloseHandle

This function closes a handle of the I^2C stream driver.

```
BOOL I2CCloseHandle(
        HANDLE hDev);
```

Parameters

hDev The I^2C device handle retrieved from CreateFile()

Return Values Returns TRUE or FALSE; if the result is TRUE, the operation is successful

11.7.2.3 I2CSetSlaveMode

This function sets the I^2C device in slave mode. This function is for back compatibility. Use **I2CEnableSlave** instead.

```
BOOL I2CSetSlaveMode(
        HANDLE hDev);
```

Parameters

hDev I^2C device handle retrieved from CreateFile()

Return Values Returns TRUE or FALSE; if the result is TRUE, the operation is successful

11.7.2.4 I2CSetMasterMode

This function sets the I^2C device in master mode. This function is for back compatibility. The default setting of driver is master.

```
BOOL I2CSetMasterMode(
        HANDLE hDev);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.5 I2CIsMaster

This function determines whether the I^2C is currently in Master mode. This function is for back compatibility.

```
BOOL I2CIsMaster(
        HANDLE hDev,
        PBOOL pbIsMaster);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbIsMaster	TRUE if the I^2C device is in master mode
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.6 I2CIsSlave

This function determines whether the I^2C is currently in Slave mode.

```
BOOL I2CIsSlave(
        HANDLE hDev,
        PBOOL pbIsSlave);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbIsSlave	TRUE if the I^2C device is in Slave mode
Return Values	Returns TRUE or FALSE. If the result is TRUE, the operation is successful

11.7.2.7 I2CGetClockRate

This function retrieves the clock rate.

```
BOOL I2CGetClockRate(
        HANDLE hDev,
        PDWORD pwClkRate);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pdwClkRate	Pointer of DWORD variable that retrieves clock rate
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.8 I2CSetClockRate

This function initializes the I^2C device with the given clock rate.

```
BOOL I2CSetClockRate(
        HANDLE hDev,
        DWORD dwClkRate);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
dwClkRate	Clock rate
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.9 I2CSetSelfAddr

This function initializes the I^2C device with the given address. The device is expected to respond when any master within the I^2C bus wish to proceed with any transfer.

```
BOOL I2CSetSelfAddr(
        HANDLE hDev,
        BYTE bySelfAddr);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
bySelfAddr	Expected I^2C device address. The valid range of address is [0x00–0x7F]
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.10 I2CGetSelfAddr

This function retrieves the address of the I^2C device.

```
BOOL I2CGetSelfAddr(
        HANDLE hDev,
        PBYTE pbySelfAddr);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbySelfAddr	Pointer to BYTE variable that retrieves I^2C device address
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.11 I2CTransfer

This function performs one or more I^2C read or write operations. **pI2CTransferBlock** contains a pointer to the first of an array of I^2C packets to be processed by the I^2C. All the required information for the I^2C operations should be contained in the array elements of pI2CPackets.

```
BOOL I2CTransfer(
        HANDLE hDev,
        PI2C_TRANSFER_BLOCK pI2CTransferBlock);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pI2CTransferBlock	
pI2CPackets	[in] Pointer to an array of packets to be transferred sequentially
iNumPackets	[in] Number of packets pointed to by pI2CPackets (the number of packets to be transferred)
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.12 I2CReset

This function performs a hardware reset. The I^2C driver maintains all the current information of the device, which includes all the initialized addresses.

```
BOOL I2CReset(
        HANDLE hDev);
```

Parameters

hDev I^2C device handle retrieved from CreateFile()

Return Values Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.13 I2CEnableSlave

This function enables a I^2C slave access from the bus. Note that this function has no effect because slave mode is not supported.

```
BOOL I2CEnableSlave(
        HANDLE hDev);
```

Parameters

hDev I^2C device handle retrieved from CreateFile()

Return Values Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.2.14 I2CDisableSlave

This function disables I^2C slave access from the bus. Note that after the I^2C slave interface disabled, I^2C slave module can be turned off.

```
BOOL I2CDisableSlave(
        HANDLE hDev);
```

Parameters

hDev I^2C device handle retrieved from CreateFile()

Return Values Returns TRUE or FALSE, if the result is TRUE, the operation is successful

11.7.3 I^2C Driver Structures

This section explains about the I^2C driver structures.

11.7.3.1 I2C_PACKET

This structure contains the information needed to write or read data using an I^2C port.

```
typedef struct {
        BYTE byRW;
        PBYTE pbyBuf;
        WORD wLen;
        LPINT lpiResult;
} I2C_PACKET, *PI2C_PACKET;
```

Parameters

byRW Determines whether the packet is a read or a write packet. Set to I2C_RW_READ for reading and I2C_RW_WRITE for writing.

pbyBuf	Pointer to a buffer of bytes. For a read operation, this is the buffer into which data is read. For a write operation, this buffer contains the data to write to the target device.The slave address must be included as part of this buffer at the right position.
wLen	If the operation is a read, wLen specifies the number of bytes to read into pbyBuf. If the operation is a write, wLen specifies the number of bytes to write from pbyBuf.
lpiResult	Pointer to an int that contains the return code from the transfer operation

11.7.3.2 I2C_TRANSFER_BLOCK

This structure contains an array of packets to be transferred using an I^2C port.

```
typedef struct {
        I2C_PACKET *pI2CPackets;
        INT32 iNumPackets;
} I2C_TRANSFER_BLOCK, *PI2C_TRANSFER_BLOCK;
```

Parameters

pI2CPackets	Pointer to an array of I2C_PACKET objects
iNumPackets	Number of I2C_PACKET objects pointed to by pI2CPackets

Chapter 12
Low-Resolution Analog-Digital Converter (LRADC) Driver

The LRADC is a multipurpose module used to measure the voltage applied to the dedicated input pins. Some of the input pins can be used to interface a resistive touchscreen, while other pins can be used for general purpose inputs. The LRADC controller is not used directly by the software.

12.1 LRADC Driver Summary

The LRADC driver can be used to measure the voltage of the General Purpose LRADC pins and to interface with a touchscreen interface, and battery interface. Thus, only one of the driver interfaces is used by the touchscreen driver. The LRADC driver interacts with the TSC to drive the LRADC.

Table 12-1 provides a summary of source code location, library dependencies and other BSP information.

Table 12-1. LRADC Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	N/A
SOC Specific Path	..PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\LRADC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\LRADC
Driver DLL	lradc.dll
SDK Library	lradcsdk_$(_SOCDIR).lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > LRADC > LRADC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_LRADC=1

12.2 Supported Functionality

The LRADC driver enables the i.MX233 EVK System to provide the following software support:

1. Configures the Touchscreen setting
2. Retrieves of the Touchscreen samples
3. Configures the general conversion setting
4. Retrieves the general purpose samples

12.3 Hardware Operation

See the *Low-Resolution ADC and Touch-Screen Interface* chapter in *i.MX233 Applications Processor Reference Manual* for hardware operation details.

12.3.1 Conflicts with Other Peripherals and Catalog Items

Because the LRADC inputs are not multiplexed with other functions, the LRADC module does not have conflict with other peripherals.

12.4 Software Operation

The LRADC device driver framework for Windows CE is a stream interface driver. A description of the stream interface driver may be found in the Windows CE Platform Builder documentation at **Developing a Device Driver** > **Windows CE Drivers** > **Stream Interface Drivers**. The LRADC stream interface driver controls the LRADC hardware. The LRADC SDK lib provides APIs for Windows CE drivers and applications. The need to access the LRADC is to use the LRADC SDK lib.

12.4.1 ADC Registry Settings

The following is the ADC registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\LRADC]
    "Dll" = "lradc.dll"
    "Prefix" = "LDC"
    "Index" = dword:1
    "Order" = dword:2
```

12.4.2 Interfacing with the LRADC Driver

This section explains how to interface with the LRADC driver.

12.4.2.1 Stream Interface

The LRADC driver is a stream interface driver, and is accessed through the file system APIs.

12.4.2.2 Using the SDK

The LRADC driver includes a wrapper library that simplifies its use. This library is the ADC SDK and is described in Section 12.7, “LRADC SDK API Reference.”

12.4.2.3 DMA Support

The LRADC driver currently does not support DMA.

12.5 Power Management

This section explains about the power management in the LRADC.

12.5.1 LDC_PowerUp

This function is not implemented for the LRADC driver.

12.5.2 LDC_PowerDown

This function is not implemented for the LRADC driver.

12.5.3 IOCTL_POWER_CAPABILITES

The power management capabilities are advertised with the power manager through this IOCTL. The LRADC module supports only two power states: D0 and D4.

12.5.4 IOCTL_POWER_SET

This function is implemented for the LRADC driver. If the clocks are disabled during the suspend (for example if the touchscreen is not a wake-up source), then the clocks are re-enabled at this time in the D0 state. If the touchscreen is not a wake-up source, then the clocks are disabled at this time in the D4 state.

12.5.5 IOCTL_POWER_GET

This IOCTL returns the current device power state. By design, the Power Manager knows the device power state of all power-manageable devices. It does not generally issue an **IOCTL_POWER_GET** call to the device unless an application calls **GetDevicePower** with the POWER_FORCE flag set.

12.6 Unit Test

Due to the heavy use of the LRADC routines by other drivers on the system, there are no additional test cases.

12.7 LRADC SDK API Reference

The following section explains about the LRADC SDK API references:

12.7.1 LRADC SDK Functions

This section explains about the functions that are in LRADC SDK.

12.7.1.1 LRADCOpenHandle

This function creates a handle to the LRADC stream driver.

```
HANDLE LRADCOpenHandle(
        LPCWSTR lpDevName
);
```

Parameters

LpDevName [in] Name of the device, for example TEXT("LDC1:")

Return Values Handle to LRADC driver which is set in this method
NULL indicates failure

12.7.1.2 LRADCCloseHandle

This function is used to closes a handle to the LRADC stream driver.

```
void LRADCCloseHandle(
        HANDLE hLRADC
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

Return Values None

12.7.1.3 LRADCConfigureChannel

This function configures a channel with the given settings.

```
BOOL LRADCConfigureChannel(
        HANDLE hLRADC,
        LRADC_CHANNEL eChannel,
        BOOL bEnableDivideByTwo,
        BOOL bEnableAccum,
        UINT8 u8NumSamples
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

eChannel [in] Identifier of the channel to configure

bEnableDivideByTwo [in] TRUE to caused the A/D converter to use its analog divide by two circuit

bEnableAccum [in] TRUE to add successive samples to the 18bit accumulator

u8NumSamples [in] Number of samples that must be converted, between 1 and 16

Return Values TRUE on success and FALSE indicates a failure

12.7.1.4 LRADCEnableInterrupt

The function enable the Interrupt of the LRADC Channel.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        LRADC_CHANNEL eChannel,
        BOOL bValue,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

eChannel [in] Identifier of the channel to configure

bValue [in] TRUE to enable, FALSE to Disable

Return Values TRUE on success and FALSE indicates a failure

12.7.1.5 LRADCClearInterruptFlag

The function clears the interrupt flag of a specified LRADC channel.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        LRADC_CHANNEL eChannel,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
eChannel	[in] Identifier of the channel to configure
Return Values	TRUE on success and FALSE indicates a failure

12.7.1.6 LRADCSetDelayTrigger

The function sets the ADC conversion sample time of the LRADC Channel.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        LRADC_DELAYTRIGGER DelayTrigger,
        UINT32 TriggerLradcs,
        UINT32 DelayTriggers,
        UINT32 LoopCount,
        UINT32 DelayCount,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
DelayTrigger	[in] Identifier of LRADC delay Triggers
TriggerLradcs	[in] The delay controller to trigger the corresponding LRADC channel
DelayTriggers	[in] The delay controller to trigger the corresponding delay channel
LoopCount	[in] The number of times this delay counter
DelayCount	[in] Delaycount of the delay channel
Return Values	TRUE on success and FALSE indicates a failure

12.7.1.7 LRADCCLearDelayChannel

The function clears the ADC conversion sample time of the LRADC Channel.

```
BOOL LRADCCLearDelayChannel(
        HANDLE hLRADC,
        LRADC_CHANNEL eChannel,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
eChannel	[in] Identifier of the channel to clear
Return Values	TRUE on success and FALSE indicates a failure

12.7.1.8 LRADCSetDelayTriggerKick

The function set the delay trigger kick of the LRADC Channel.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        LRADC_DELAYTRIGGER DelayTrigger,
        BOOL bValue,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
DelayTrigger	[in] Identifier of LRADC delay Triggers
bValue	[in] TRUE to enable, FALSE for Disable
Return Values	TRUE on success and FALSE indicates a failure

12.7.1.9 LRADCGetAccumValue

The function gets the conversion value of a specified LRADC channel.

```
UINT16 LRADCGetAccumValue(
        HANDLE hLRADC,
        LRADC_CHANNEL Channel,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
eChannel	[in] Identifier of the channel to configure
Return Values	Accumulator value of the channel

12.7.1.10 LRADCEnableBatteryMeasurement

The function enable the Interrupt of the LRADC Channel.

```
BOOL LRADCEnableBatteryMeasurement(
        HANDLE hLRADC,
        LRADC_DELAYTRIGGER eTrigger,
        UINT32 TriggerLradcs,
        LRADC_BATTERYMODE eBatteryMode,
);
```

Parameters

hLRADC	[in] Handle to configure retrieved from LRADCOpenHandle
eTrigger	[in] Identifier of LRADC delay Triggers
riggerLradcs	[in] Specifies the sampling interval for the Battery value update
eBatteryMode	[in] Specifies the Battery mode setup
Return Values	Return 0 If the operation is successful otherwise returns error value failure

12.7.1.11 LRADCEnableDieMeasurement

The function enables the LRADC channel for die temperature measurement.

```
BOOL LRADCEnableDieMeasurement(
        HANDLE hLRADC,
        );
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

Return Values Return die temperature.

12.7.1.12 LRADCClearAccum

The function Clears the Accum Value of the specified LRADC channel.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        LRADC_CHANNEL Channel,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

eChannel [in] Identifier of the channel to configure

Return Values TRUE on success and FALSE indicates a failure

12.7.1.13 LRADCEnableTouchDetect

The function set or clear the TOUCH_DETECT_ENABLE in HW_LRADC_CTRL0 Register.

```
BOOL LRADCEnableTouchDetect(
        HANDLE hLRADC,
        BOOL bValue,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

bValue [in] TRUE for set, FALSE for clear

Return Values TRUE on success and FALSE indicates a failure

12.7.1.14 LRADCGetTouchDetect

The function Read the TOUCH_DETECT_RAW bit of HW_LRADC_STATUS register.

```
BOOL LRADCGetTouchDetect(
        HANDLE hLRADC,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

Return Values TRUE on success and FALSE indicates a failure

12.7.1.15 LRADCEnableTouchDetectInterrupt

The function set or clear the TOUCH_DETECT_IRQ_EN in HW_LRADC_CTRL1 Register.

```
BOOL LRADCEnableTouchDetectInterrupt(
        HANDLE hLRADC,
        BOOL bValue,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

bValue [in] TRUE to enable, FALSE for Disable

Return Values TRUE on success and FALSE indicates a failure

12.7.1.16 LRADCSetAnalogPowerUp

The function set or clear the ADC analog power up.

```
BOOL LRADCEnableInterrupt(
        HANDLE hLRADC,
        BOOL bValue,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

bValue [in] TRUE to enable, FALSE for Clear

Return Values TRUE on success and FALSE indicates a failure

12.7.1.17 LRADCClearTouchDetect

The function clear the touch detect status.

```
BOOL LRADCClearTouchDetect(
        HANDLE hLRADC,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

Return Values TRUE on success and FALSE indicates a failure

12.7.1.18 LRADCEnableTouchDetectXDrive

The function enable or disable the X Channels in HW_LRADC_CTRL0 Register.

```
BOOL LRADCEnableTouchDetectXDrive(
        HANDLE hLRADC,
        BOOL bValue,
);
```

Parameters

hLRADC [in] The LRADC device handle retrieved from LRADCOpenHandle

bValue [in] TRUE to enable, FALSE for Disable

Return Values TRUE on success and FALSE indicates a failure

12.7.1.19 LRADCEnableTouchDetectYDrive

The function enable or disable the Y Channels in HW_LRADC_CTRL0 Register.

```
BOOL LRADCEnableTouchDetectYDrive(
        HANDLE hLRADC,
        BOOL bValue,
);
```

Parameters

hLRADC	[in] The LRADC device handle retrieved from LRADCOpenHandle
bValue	[in] TRUE to enable, FALSE for Disable
Return Values	TRUE on success and FALSE indicates a failure

Chapter 13
NAND Flash Driver

The NAND flash driver provides the functionality of NAND storage accessing. The flash driver follows Windows CE 6.0 R2 flash driver having module device driver (MDD) and platform-dependent driver (PDD) model.

13.1 Flash Driver Summary

Windows CE provides driver support for flash media devices using MDD or PDD architecture. The MDD allows NAND flash storage to be exposed as a block driver that is accessed by file system. The PDD wraps FMD layer (flash driver model before R2) as a stream interface called by MDD. The FMD software layer ported to the i.MX NAND flash controller is responsible for the actual communication with the corresponding NAND flash devices.

The flash driver supports both SLC and MLC NAND flash devices. As for page size, 512 byte (small page size) is not supported.

Table 13-1 provides a summary of source code location, library dependencies and other BSP information.

Table 13-1. Flash Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\NAND
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\MEDIA\NAND
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\NANDFMD ..\PLATFORM\<*Target Platform*>\SRC\COMMON\NANDFMD
Driver DLL	flashpdd_nand.dll
SDK Library	N/A
Catalog Item	Device Drivers > Storage Devices > MSFlash Drivers > Flash MDD Third Party > BSP > Freescale i.MX233 EVK: ARMV4I > Storage Drivers > Flash > NAND Flash
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NONAND_FMD=

13.2 Supported Functionality

The flash driver enables the EVK System to provide the following software and hardware support:

1. Supports the Windows CE MDD or PDD interface
2. Supports both MLC and SLC NAND
3. Supports both 2 Kbyte and 4 Kbyte page size NAND
4. Boot time dynamic detection of installed flash module
5. Supports MLC NAND Flash MT29F8G08MAD as default
6. Supports MLC NAND Flash MT29F16G08MAD
7. Supports MLC NAND Flash MT29F32G08QAAWP
8. Supports SLC NAND Flash MT29F16G08DAA

13.3 Hardware Operation

See the chapter on the NAND Flash Controller (GPMI) in the *i.MX233 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

13.3.1 Conflicts with Other Peripherals and Catalog Items

This section explains the conflicts with other peripherals and catalog items.

13.3.1.1 Conflicts with SoC Peripherals

No conflicts.

13.4 Software Operation

The development concepts for flash media drivers are described in the Windows CE 6.0 Help Documentation in the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Flash Drivers.

The flash driver supported in the i.MX BSP implements the required PDD functions for interfacing to NAND Flash devices.

13.4.1 MDD and PDD Layer Overview

The Microsoft Windows Embedded CE 6.0 flash driver component contains two components, the MDD and the PDD.

The flash driver MDD is responsible for actions such as handling wear-leveling, writing sector transactions, translating logical sectors to physical sectors, and performing compaction. The flash MDD can operate regardless of the type of flash media, allowing it to support single-level cell (SLC) NAND, multi-level cell (MLC) NAND, and NOR media. The OS provides the MDD component.

The flash driver PDD is responsible for interacting with the flash hardware, and contains the basic functions necessary to access a physical flash. Also, the PDD exposes a stream interface, where the user can implement the PDD IOCTLs to meet your specific hardware needs. The PDD component is platform specific, and the Freescale flash driver provides the functionality of the PDD component.

The block diagram shown in Figure 13-1 describes the high level architecture and basic interactions of the i.MX NAND driver implementation. The i.MX flash driver PDD consists of three major components:

- Common Logical Layer—Contains logical part of the PDD layer, including parameter check, memory management, boot time dynamic detection of installed flash module, algorithm for using multiple NAND chips, and so on. This layer is shared by all platforms.
- SOC Operation Layer—Contains pure hardware operations, including sector reading, sector writing, block erasing, and so on. No additional logic is present in this layer, except some simple logic for doing hardware operations. This layer is SOC specific.
- BSP Configuration Layer—This component is used to report flash chip properties to common logical layer. No algorithm and no hardware operations are needed in this layer. Only report the reality situation of the flash property on board. This layer is board specific.

The i.MX flash driver currently supports a limited number of commercially available flash modules. However, the i.MX flash driver software architecture supports new flash modules also. The i.MX flash driver must be modified to support new flash modules, by changing the BSP not to support the current flash module.

The i.MX flash driver is table driven. That is, by appropriately modifying the data structures, the flash driver can be reconfigured to support a different flash module. No other source code changes are required.

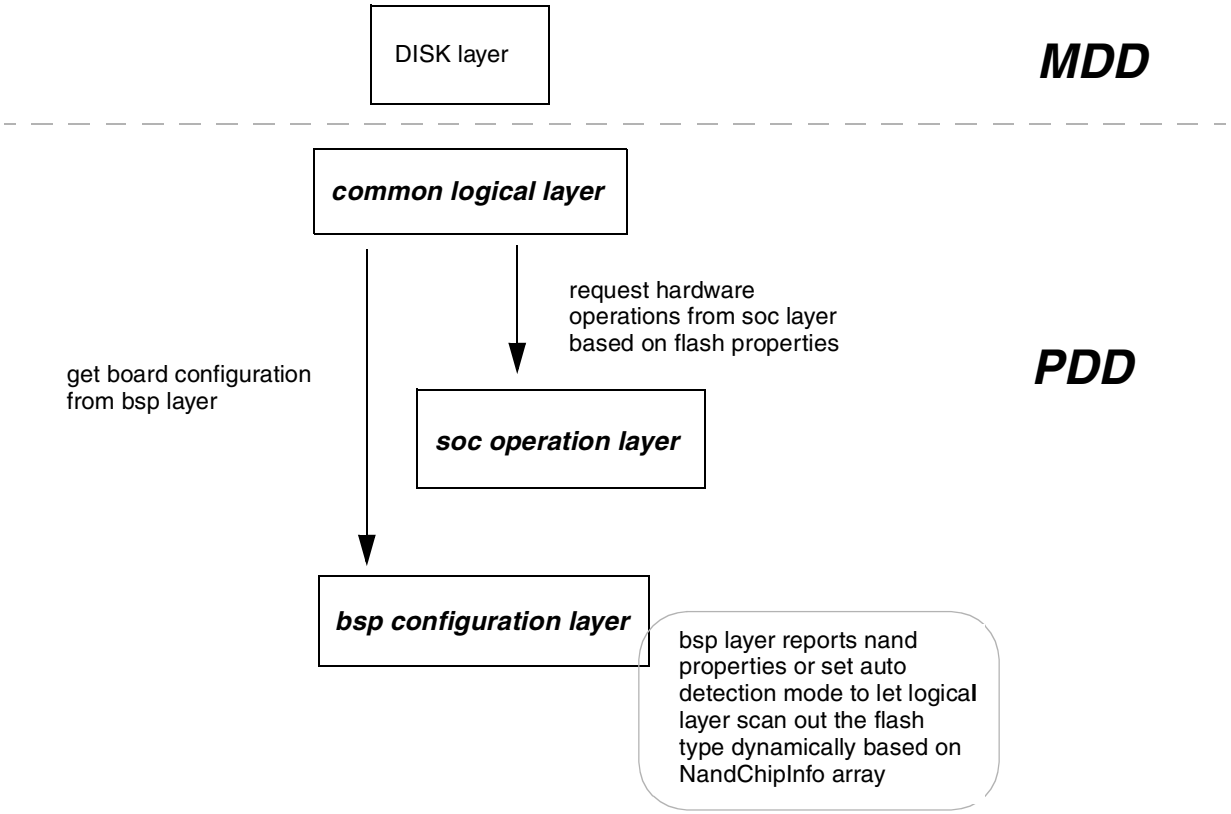

Figure 13-1. PDD Layer Block Diagram

13.4.2 Data Structures

The flash modules vary between manufacturers, and even between process-technologies or product revisions by the same manufacturer. Each module is different, and the flash driver must change to support these new modules. A number of definitions are used to describe flash module characteristics and include the following:

- Bad block mark
- Block size
- Page size
- Command set

The manufacturer's data sheet describes each of these definitions in detail for a particular flash module. The manufacturer's data sheet and these definitions are very important to understand when adding a new flash support to the i.MX flash driver.

13.4.2.1 NANDTiming

The following code is the NANDTiming structure:

```
typedef struct _NANDTiming{
      BYTE DataSetup;
      BYTE DataHold;
      BYTE AddressSetup;
      BYTE DataSample;
} NANDTiming, *PNANDTiming;
```

Table 13-2 lists the members in NANDTiming structure.

Table 13-2. NANDTiming Structure Members

Member	Description
DataSetup	Data setup time in microsecond unit.
DataHold	Data hold time in microsecond unit.
AddressSetup	Address setup time in microsecond unit.
DataSample	Data sample time in microsecond unit.

13.4.2.2 NandChipInfo

The following code is the NandChipInfo structure:

```
typedef struct _NandChipInfo{
      FlashInfo fi;
      BYTE NANDCode[NANDID_LENGTH];
      BYTE NumBlockCycles;
      BYTE ChipAddrCycleNum;
      BYTE DataWidth;
      BYTE BBMarkNum;
      BYTE BBMarkPage[MAX_MARK_NUM];
      BYTE StatusBusyBit;
      BYTE StatusErrorBit;
      WORD SpareDataLength;
      BYTE CmdReadStatus;
      BYTE CmdRead1;
      BYTE CmdRead2;
      BYTE CmdReadId;
      BYTE CmdReset;
      BYTE CmdWrite1;
      BYTE CmdWrite2;
      BYTE CmdErase1;
      BYTE CmdErase2;
      NANDTiming timings;
} NandChipInfo, *PNandChipInfo;
```

Table 13-3 lists the members in NandChipInfo structure.

Table 13-3. NandChipInfo Structure Members

Member	Description
fi	Flash information structure, see details in MSDN for Windows CE6.
NANDCode	The first 4 bytes of NAND flash ID.
NumBlockCycles	Number of row address cycles.
ChipAddrCycleNum	Number of row and column address cycles.
DataWidth	Bit number of NAND flash, it should be 8bits or 16bits.
BBMarkNum	Number of pages, defined by manufacturer, that is used to indicate initial bad block during manufacturing.
BBMarkPage	An array that indicates which pages are used to indicate initial bad block during manufacturing.
StatusBusyBit	Bit number in status byte to indicate BUSY or IDLE status of NAND flash status.
StatusErrorBit	Bit number in status byte to indicate PASS or FAIL status of NAND flash operation.
SpareDataLength	Number of bytes in spare area per page.
CmdReadStatus	Command used to read NAND flash status.
CmdRead1	Command used as initial command for reading operation.
CmdRead2	Command used as start command for reading operation.
CmdReadId	Command used to read NAND flash ID.
CmdReset	Command used to reset NAND flash.
CmdWrite1	Command used as initial command for writing operation.
CmdWrite2	Command used as start command for writing operation.
CmdErase1	Command used as initial command for erasing operation.
CmdErase2	Command used as start command for erasing operation.
timings	Parameters to indication NAND flash timing information.

13.4.3 Adding New Flash Configurations

The i.MX flash driver is table driven. That is, by appropriately modifying the data structures, the flash driver can be reconfigured to support a different flash module. No other source code changes are required.The Freescale flash driver supports boot time dynamic detection of previously verified flash modules. The flash driver dynamically detects the flash modules listed in Section 13.2, “Supported Functionality.” At boot time, the flash driver query about the installed flash module, and loads the appropriate flash module configuration to support the specific flash module.

To add an unsupported flash module to the BSP, the flash driver must be configured to support a new NAND module. After a new flash definition configuration is added, the flash driver will automatically recognize the new flash module at boot time.

To add a new flash module support, a new `NandChipInfo` data structure member must be added to the `NANDTYPES.h` header file. Then the flash driver must be re-compiled and the platform must be rebuilt. The `NANDTYPES.h` header file is located in the following directory:

`\WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\NAND\INC\NANDTYPES.h`

NOTE

The flash driver currently supports 2K+64B page size, 4K+128B page size, and 4K+218B page size with 8 bit ECC. The table configuration method can be used to support these common flash memory types.

13.4.4 Registry Settings

The registry keys implemented for the flash driver provide basic support for loading and configuring the NAND as a file system mount. Many configuration options are available and are discussed in Windows CE 6.0 Help documentation at the following location:

Windows Embedded CE features > **File Systems and Storage Management** > **Storage Management** > **Storage Manager Registry Settings**

As default, the NAND disk is automatically formatted and a partition is created, when no NAND partition is found while booting up. The functionality is implemented by specifying following items:

```
"AutoPart"=dword:1
"AutoFormat"=dword:1
```

The two items can be deleted to disable the functionality.

13.4.5 DMA Support

The flash driver supports DMA mode.

13.4.6 Power Management

The flash driver handles power requests in MDD layer by default.

13.5 Unit Test

The flash driver is tested using the Windows CE 6.0 Test Kit and additional system used cases. This section describes the test scenarios that are used to verify the operation of the flash driver.

13.5.1 CETK Testing

This section about the CETK testing used to verify the operation of the flash driver.

NOTE

Depending on the state of the NAND flash memory, it may be necessary to format and partition the NAND device using Storage Manager prior to running the CETK tests that do not reformat the device automatically.

Table 13-4 lists the CEKT tests used for the flash driver.

Table 13-4. CEKT Tests

CETK Test	Command Line
File System Driver Test	tux -o -d fsdtst -c "-p MSFlash -z"
Flash Memory Read/Write and Performance Test	tux -o -d flshwear -c"/profile MSFlash /store /flash"
Storage Device Block Diver API Test	tux -o -d disktest -c"/profile MSFlash /zorch /part /sectors 256"
Storage Device Block Diver Benchmark Test	tux -o -d rw_all -c"/profile MSFlash /zorch /part"
Storage Device Block Diver Read/Write Test	tux -o -d rwtest -c"/profile MSFlash /zorch /part"

13.5.2 System Testing

The following system tests verify the operation of the flash driver:

- Use the **Start** > **Settings** > **Control Panel** > **Storage Manager** to format and create partitions on the mounted NAND device
- Establish ActiveSync connection over USB and transfer files to or from the NAND storage
- Write media files to NAND storage. Use Windows Media Player to playback media files from NAND storage

Chapter 14
NAND Redundant Boot

Redundant boot supported from NAND includes the following components:

- Boot Image checking tool
- Boot Image updating tool

Boot Image checking tool is used for checking boot streams integrity every time when system boots up; Boot Image updating tool is used for updating image. If the update fails, the checking tool can easily restore the image when next time boots up. These tools cannot run simultaneously to prevent boot stream corruption.

14.1 NAND Redundant Boot Summary

Table 14-1 provides a summary of source code location, library dependencies and other BSP information.

Table 14-1. NAND Redundant Boot Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX233-EVK
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\COMMON\NANDBOOTBURNER ..\PLATFORM\<*Target Platform*>\SRC\APP\UpdateSB
Driver DLL	N/A
SDK Library	N/A
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	N/A

14.2 Supported Functionality

The NAND Redundant Boot enables the system to provide the following software and hardware support:

1. Supports updating image from certain location on the device.
2. Supports restoring backup image when the update fails.
3. Supports updating backup image when user confirms the updated image works well.

14.3 Hardware Operation

This section explains about the hardware operations.

14.3.1 Conflicts with Other Peripherals and Catalog Items

No conflict.

14.4 Software Operation

Figure 14-1shows the Boot Image updating tool.

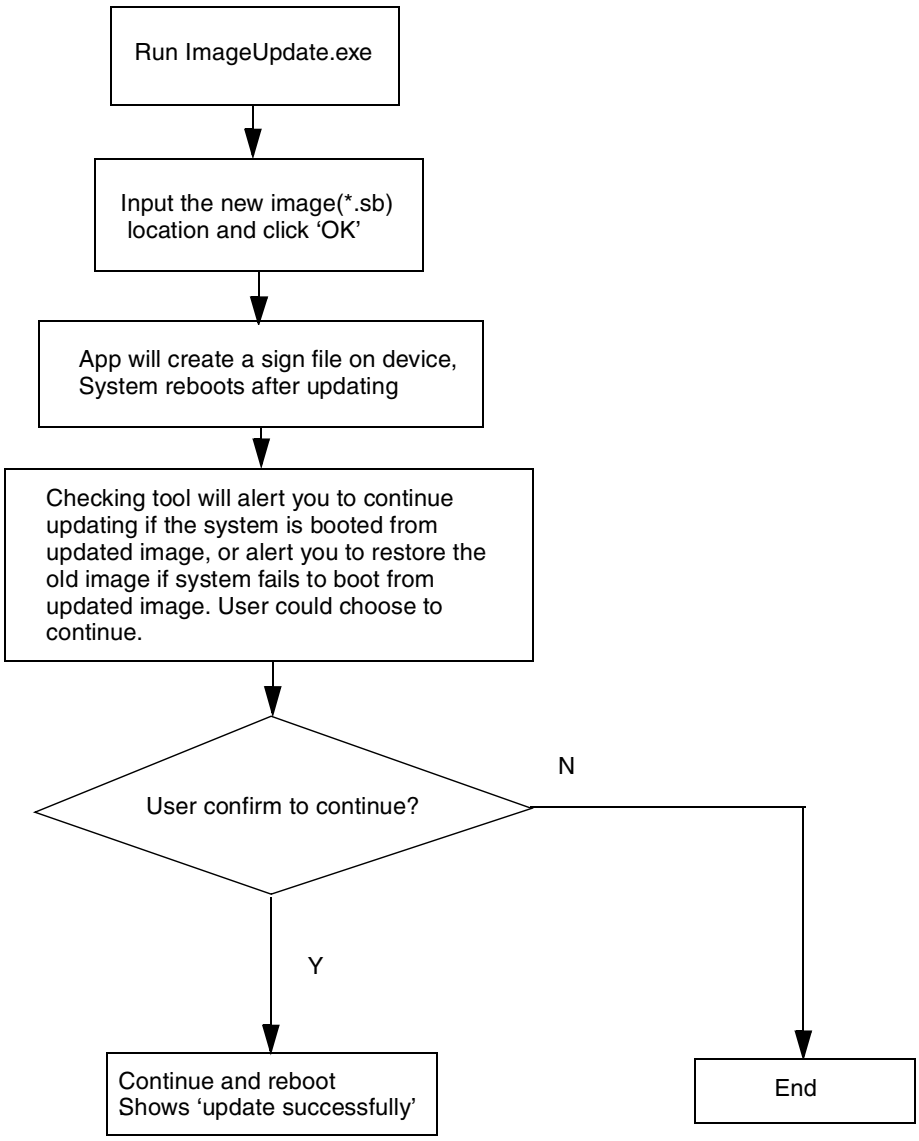

Figure 14-1. Image Updating Work Flow

Figure 14-2 shows the Boot Image checking tool.

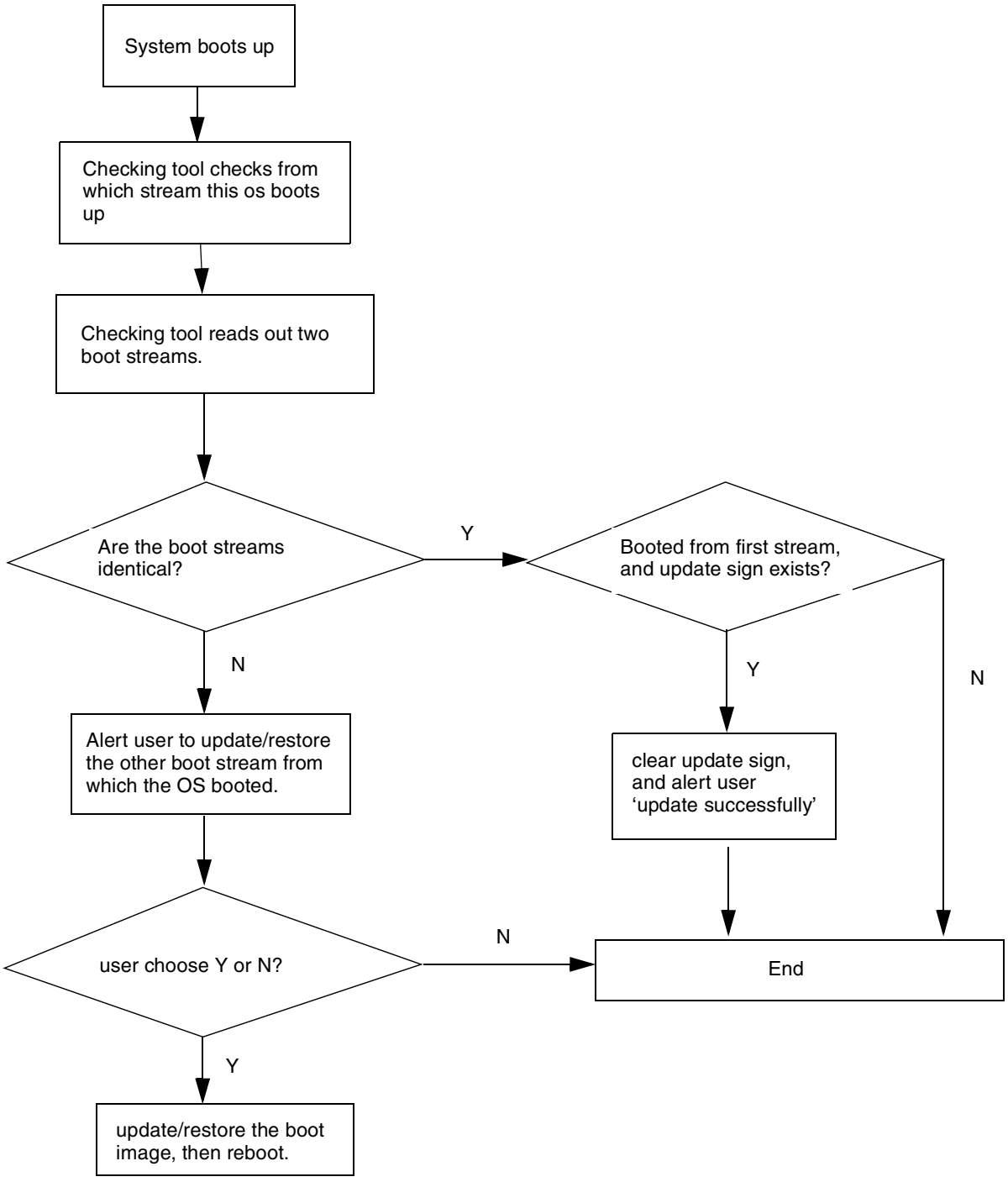

Figure 14-2. Image Checking Work Flow

14.5 Unit Test

The following section describes the testing update and restore functionality:

14.5.1 Testing Update Functionality

Use the following steps to test the update functionality:

1. Run ImageUpdate.exe
2. Select the new image(*.sb)
3. Click **OK**.
4. Image is updated and system reboots after update.
5. Make sure this startup is from new image.
6. A continue update message appears asking the user to confirm the update.
7. Click **YES** to continue.
8. Image will be updated and system will reboot after this updating.
9. After this startup, a **update successfully** message appears.

14.5.2 Testing Restore Functionality

Use the following steps to test the restore functionality:

1. Run ImageUpdate.exe
2. Select the new image(*.sb)
3. Click **OK**.
4. Image is updated and system reboots after update.
5. Make sure this startup is from new image.
6. There will be a message to ask user to continue updating.
7. Click **NO.**
8. Power off the device, then power on.
9. After this startup, there will be a message to ask user to recover.
10. Click **YES** to continue.
11. Image is recovered and system reboots.
12. Make sure this startup is from old image.

Chapter 15
Serial Driver

The serial driver interfaces the low level serial driver hardware to the Windows CE serial subsystem.

15.1 Serial Driver Summary

The serial port driver is implemented as a stream interface driver and supports all the standard I/O control codes and entry points. The serial port driver handles all the internal UARTs except UART1 which is used for debugging. In the BSP implementation, the hardware-specific code that corresponds to the serial port driver lower layer is implemented as the platform-dependent driver (PDD). This PDD is linked with Microsoft-provided public serial MDD library (com_mdd2.lib) to form the whole serial port driver. Table 15-1 provides a summary of source code location, library dependencies and other BSP information.

Table 15-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\MX233_FSL_V2\SERIAL
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\DRIVERS\SERIAL
Driver DLL	csp_serial.dll
SDK Library	N\A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Serial > UART2
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SERIAL_UART2 =1

15.2 Supported Functionality

The serial port driver enables the hardware system to provide the following support:

1. Conforms to RS232 protocol standard
2. Supports RTS/CTS hardware flow control function
3. Supports parity check and optional stop bit
4. Supports power management mode full on/full off
5. Supports baud rate up to 3.25 Mbps

15.3 Hardware Operation

See the chapter *Multimedia Applications Processor Reference Manual* for detailed operation and programming information on UART.

15.3.1 Conflicts with Other Peripherals and Catalog Items

The following section explains serial driver conflicts with other peripherals and catalog items.

15.3.1.1 Conflicts with SoC Peripherals

All the pins of UART can be configured for alternate functionality (GPMI, IR, SSP) using the i.MX233 IOMUX. The configuration is specified by BSP serial driver. Changing this configuration would result in a conflict and prevent proper operation of the UART.

15.3.1.2 Conflicts with Board Peripherals

No conflicts.

15.3.2 Known Issues

DMA mode is not supported due to the DMA module limitation. DMA engine will fail to flush the last bytes less than 4 bytes into RX buffer.

15.4 Software Operation

The serial driver follows the Microsoft-recommended architecture for serial drivers. The details of this architecture and its operation can be found in the Platform Builder Help documentation at the following location:

Developing a Device Driver > Windows CE Drivers > Serial Drivers > Serial Driver Development Concepts.

15.4.1 Registry Settings

This section explains about the registry settings used to load the serial driver:

15.4.1.1 i.MX233 Registry Settings

The following registry keys are required to load the serial driver:

```
IF BSP_SERIAL_UART2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:8006C000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
```

```
   "Index"=dword:2
   "Order"=dword:3
   "useDMA"=dword:0
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="i.MX233 COM2 UNIMODEM"
   "DevConfig"=hex: 10,00, 00,00, 05,00,00,00, 10,01,00,00, 00,4B,00,00, 00,00, 08, 00, 00,
00,00,00,00
ENDIF ; BSP_SERIAL_UART2
```

15.4.2 Power Management

The serial driver supports full on/full off power management mode through PowerUp( ) and PowerDown( ) functions.

15.5 Unit Test

The serial driver is tested using the Serial Port Driver Test and Serial Communications Test included as a part of the CETK. The Serial Port Test assesses whether the driver supports configurable device parameters such as baud rate and data bits. The test also assesses additional functionality such as COM port events, escape functions and time-outs.

15.5.1 Unit Test Hardware

- i.MX233 EVK board

15.5.2 Unit Test Software

Table 15-2 lists the required software to run the unit tests.

Table 15-2. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
SerDrvBvt.dll	Test.dll file for Serial Port Driver Test

15.5.3 Building the Unit Tests

The serial port driver tests come pre-built as part of the CETK. No steps are required to build these tests. The Pserial.dll file can be found alongside the other required CETK files in the following location:

```
[Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4i
```

15.5.4 Running the Unit Tests

The Serial Port Driver Test executes the `tux -o -d serdrvbvt` command line on default execution.

For detailed information on the Serial Port Tests, see

Debugging and Testing > Tools for Debugging and Testing > Windows CE Test Kit > CETK Tests > Serial Port Driver Test > Serial Port Driver Test Cases in the Platform Builder Help.

The Serial Port Tests are designed to test that the serial port driver works properly and the API behaves correctly, and it should be pass all the test cases. Table 15-3 describes the Serial Port driver test cases.

Table 15-3. Serial Port Driver Test Cases

Test Case	Description
1001	Configures the port and writes data to the port at all possible baud rates, data bits, parities, and stop bits. This test fails if it cannot send data on the port with a particular configuration.
1002	Tests the **SetCommEvent** and **GetCommEvent** functions. This test fails if the driver does not properly support the **SetCommEvent** or **GetCommEvent** functions.
1003	Tests the **EscapeCommFunction** function. This test fails if the driver does not support one of the Microsoft Win32 **EscapeCommFunction** functions.
1004	Tests the **WaitCommEvent** function on the EV_TXEMPTY event. The test creates a thread to send data and waits for the EV_TXEMPTY event to occur when the thread finishes sending data. This test fails if the **WaitCommEvent** function behaves improperly or if the EV_TXEMPTY event does not signal appropriately.
1005	Tests the **SetCommBreak** and **ClearCommBreak** functions. This test fails if the driver does not properly support the **SetCommBreak** or **ClearCommBreak** functions.
1006	Makes the **WaitCommEvent** function return a value when the handle for the current COM port is cleared. This test fails if the **WaitCommEvent** function behaves improperly.
1007	Makes the **WaitCommEvent** function return a value when the handle for the current COM port is closed. This test fails if the **WaitCommEvent** function behaves improperly.
1008	Tests the **SetCommTimeouts** function and verifies that the **ReadFile** function properly times out when no data is received. This test fails if the COM timeouts do not function correctly.
1009	Verifies that previous Device Control Block (DCB) settings are preserved when the **SetCommState** function call fails with DCB settings that are not valid. This test fails if the serial port driver does not keep previous DCB settings when DCB settings that are not valid are passed to the driver.

15.6 Serial Driver API Reference

The detailed reference information for the serial driver may be found in the Platform Builder Help at the following location:

Developing a Device Driver > Windows CE Drivers > Serial Port Drivers > Serial Port Driver Reference

15.6.1 Serial PDD Functions

Table 15-4 shows a mapping of Serial PDD functions to the functions used in the serial driver.

Table 15-4. Serial PDD Functions

PDD Function Pointer	Serial Driver Function
HWInit	SerSerialInit
HWPostInit	SerPostInit
HWDeinit	SerDeinit
HWOpen	SerOpen
HWClose	SerClose
HWGetIntrType	SL_GetIntrType
HWRxIntrHandler	SL_RxIntrHandler
HWTxIntrHandler	SL_TxIntrHandler
HWModemIntrHandler	SL_ModemIntrHandler
HWLineIntrHandler	SL_LineIntrHandler
HWGetRxBufferSize	SL_GetRxBufferSize
HWPowerOff	SerPowerOff
HWPowerOn	SerPowerOn
HWClearDTR	SL_ClearDTR
HWSetDTR	SL_SetDTR
HWClearRTS	SL_ClearRTS
HWSetRTS	SL_SetRTS
HWEnableIR	SerEnableIR
HWDisableIR	SerDisableIR
HWClearBreak	SL_ClearBreak
HWSetBreak	SL_SetBreak
HWXmitComChar	SL_XmitComChar
HWGetStatus	SL_GetStatus
HWReset	SL_Reset
HWGetModemStatus	SL_GetModemStatus
HWGetCommProperties	SerGetCommProperties
HWPurgeComm	SL_PurgeComm
HWSetDCB	SL_SetDCB
HWSetCommTimeouts	SL_SetCommTimeouts

15.6.2 Serial Driver Structures

This section explains about the serial driver structures.

15.6.2.1 UART_INFO

This structure contains information about the UART Module.

```
typedef struct {
    volatile PCSP_UART_REG    pUartReg;
    ULONG    sUSR1;
    ULONG    sUSR2;
    BOOL    bDSR;
    uartType_c    UartType;
    ULONG    ulDiscard;
    BOOL    UseIrDA;
    ULONG    HwAddr;
    EVENT_FUNC    EventCallback;
    PVOID    pMDDContext;
    DCB    dcb
    COMMTIMEOUTS    CommTimeouts;
    PLOOKUP_TBL    pBaudTable;
    ULONG    DroppedBytes;
    HANDLE    FlushDone;
    BOOL    CTSFlowOff;
    BOOL    DSRFlowOff;
    BOOL    AddTXIntr;
    COMSTAT    Status;
    ULONG    CommErrors;
    ULONG    ModemStatus;
    CRITICAL_SECTION    TransmitCritSec;
    CRITICAL_SECTION    RegCritSec
    ULONG    ChipID;
 } UART_INFO, * PUART_INFO;
```

Parameters

pUartReg	Pointer to UART Hardware registers
sUSR1	This value contains the UART status register
sUSR2	This value contains the UART status register
bDSR	This boolean value keeps the DSR state
UartType	This value contains the type of UART like DCE or DTE
UlDiscard	This is used to discard the echo characters in IrDa Mode
UseIrDA	This boolean value determines the driver is in IR mode or not
HwAddr	This value contains the hardware address of the UART Module
EventCallback	This is a callback to the Model Device Driver
pMDDContext	This contains the context of the UART, which is the first parameter to the callback function
dcb	This value contains the copy of Device Control Block

CommTimeouts	This contains the copy of CommTimeouts structure used to get and set the time-out parameters for a communication device
pBaudTable	Pointer to baud rate table
DroppedBytes	This value contains the number of bytes dropped
FlushDone	Handle to the flush done event
CTSFlowOff	This boolean value is used to store the CTS flow control state
DSRFlowOff	This boolean value is used to Store the DSR flow control state
AddTXIntr	This boolean value is used to fake a Tx interrupt
Status	This value contains the comm status
CommErrors	This value contains Win32 comm error status
ModemStatus	This value shows the Win32 Modem status
TransmitCritSec	This value is used as Critical Section for UART registers
RegCritSec	This value is used as Critical Section for UART
ChipID	This value contains Chip identifier (CHIP_ID_16550 or CHIP_ID_16450)

15.6.2.2 SER_INFO

This is a private structure contains the information about the serial.

```
typedef struct __SER_INFO {
    UART_INFO    uart_info;
    BOOL    fIRMode;
    DWORD    dwDevIndex;
    DWORD    dwIOBase;
    DWORD    dwIOLen;
    PCSP_UART_REG pBaseAddress;
    UINT8    cOpenCount;
    COMMPROP    CommProp;
    PHWOBJ    pHWObj;
    BOOL        useDMA;
    DDK_DMA_REQ     SerialDmaReqTx;
    DDK_DMA_REQ     SerialDmaReqRx;
    PHYSICAL_ADDRESS SerialPhysTxDMABufferAddr;
    PHYSICAL_ADDRESS SerialPhysRxDMABufferAddr;
    PBYTE       pSerialVirtTxDMABufferAddr;
    PBYTE       pSerialVirtRxDMABufferAddr;
    UINT8       SerialDmaChanRx;
    UINT8       SerialDmaChanTx;
    UINT8       currRxDmaBufId;
    UINT8       currTxDmaBufId;
    UINT        dmaRxStartIdx;
    UINT        availRxByteCount;
    UINT32      awaitingTxDMACompBmp;
    UINT32      dmaTxBufFirstUseBmp;
    UINT16      rxDMABufSize;
    UINT16      txDMABufSize;
} SER_INFO, *PSER_INFO;
```

Parameters

uart_info	This structure contains information about UART
fIRMode	This boolean value determines the module is FIR or serial
dwDevIndex	This static value contains the device index value which is read from registry
dwIOBase	This static value contains the I/O Base address of UART module which is read from registry
dwIOLen	This static value contains the I/O length of UART Module which is read from registry
pBaseAddress	Pointer to the start address of the UART registers mapped
cOpenCount	Contains count of the concurrent open
CommProp	Pointer to CommProp structure
pHWObj	Pointer to PDDs HWObj structure
useDMA	This boolean flag indicates if SDMA is to be used for transfers through this UART
SerialDmaReqTx	SDMA request line for Tx
SerialDmaReqRx	SDMA request line for Rx
SerialPhysTxDMABufferAddr	Physical address of Tx SDMA address
SerialPhysRxDMABufferAddr	Physical address of Rx SDMA address
pSerialVirtTxDMABufferAddr	Virtual address of Tx SDMA address
pSerialVirtRxDMABufferAddr	Virtual address of Rx SDMA address.
SerialDmaChanRx	SDMA virtual channel indices for Rx
SerialDmaChanTx	SDMA virtual channel indices for Tx
currRxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Rx SDMA buffer descriptor chains
currTxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Tx SDMA buffer descriptor chains
dmaRxStartIdx	Keeps the start index of byte to be delivered to MDD for Read
availRxByteCount	This variable keeps the remaining bytes in the Rx SDMA buffer
awaitingTxDMACompBmp	Indicates if an SDMA request is in progress on Tx SDMA buffer descriptor
dmaTxBufFirstUseBmp	Indicator for first time use of a Tx SDMA buffer descriptor
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size

Chapter 16
Secure Digital Host Controller (SDHC) Driver

The SDHC module supports MMC, SD cards and Secure Digital I/O. The SDHC driver provides the interface between the Microsoft SD Bus driver and the SSP hardware.

16.1 SDHC Driver Summary

Table 16-1 provides a summary of source code location, library dependencies and other BSP information.

Table 16-1. eSDHC Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\SDHC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SDHC
Driver DLL	sdhc.dll
SDK Library	sdhc_<*Target SOC*>.lib, sdcardlib.lib, sdhclib.lib, sdbus.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > SD Controller > SSP1 SDHC
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	N/A

16.2 Supported Functionality

The SDHC driver enables the EVK System to provide the following software and hardware support:

1. Supports the Synchronous Serial Ports(SSP) Controller
2. Designed and implemented as close as possible to Standard Host Controller Driver in CE 6.0 R2
3. Compliant with the SDBUS2 driver provided in CE 6.0 R2
4. Supports Fast Path
5. Supports DMA mode of data transfers
6. Supports SD, SD High Capacity (up to spec v2.1), MMC (up to spec v4.3), and SDIO cards (up to spec v2.0). High capacity MMC cards are not supported because SDBUS2 in CE 6.0 R2 does not support these cards
7. One host supports only one card to be connected to it

8. DLL supports multiple instances of the SSP controller
9. Supports the configuration of the block sizes from 1-4096 bytes in single and multi-block modes
10. Support insertion and removal of card, even when system is suspended; when the system resumes, the card (if present) is remounted
11. Supports the write protect switch on SD cards
12. Supports MMC cards in 1-bit mode and SD/SDIO cards in 4-bit modes due to limitation in SDBUS2 in CE 6.0 R2

16.3 Hardware Operation

See the i.MX233 *Multimedia Applications Processor Reference Manual* for detailed operation and programming information on SSP.

16.3.1 Conflicts with Other Peripherals and Catalog Options

This section explains SDHC driver conflicts with other peripherals and catalog options.

16.3.1.1 Conflicts with SoC Peripherals

No conflicts.

16.3.1.2 Conflicts with Other EVK Peripherals

On the i.MX233-EVK platform, the SDHC conflicts with Ethernet controller on SSP1 port. If Ethernet KITL is detected, then SDHC driver does not load. The USB RNDIS KITL can be used instead, so that KITL and SDHC driver can work simultaneously.

16.4 Software Operation

The SDHC driver follows the Microsoft-recommended architecture (standard host controller driver) for Secure Digital Host Controller drivers, whenever possible. The details of this architecture and its operation can be found in the Platform Builder Help under the heading **Secure Digital Card Driver Development Concepts**, or in the online documentation at the following URL:
http://msdn2.microsoft.com/en-us/library/aa925967.aspx

16.4.1 Required Catalog Items

The following are the required catalog items:

16.4.1.1 SD and MMC Support

Catalog > Device Drivers > SDIO > SD Memory

Additionally, since eSDHC driver supports high capacity cards, it is necessary to define IMGSDBUS2 variable in the workspace. Both SYSGEN_SD_MEMORY and IMGSDBUS2 are set by default in the BSP workspace.

16.4.2 SDHC Registry Settings

This section explains about SDHC registry settings.

16.4.2.1 i.MX233 SDHC Registry Settings

The following registry keys are required to load SDHC driver:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SDHC]
    "Order"=dword:19
    "Dll"="sdhc.dll"
    "Prefix"="SHC"
    "Index"=dword:1

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\MMC]
    "Name"="MMC Card"
    "Folder"="MMCMemory"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\SDMemory]
    "Name"="SD Memory Card"
    "Folder"="SDMemory"
```

16.4.3 DMA Support

DMA mode is supported by the SDHC driver. The driver uses the APBH DMA, which has a special SSP1 DMA channel.

16.4.4 Power Management

The SHC_powerUp and SHC_PowerDown APIs are the entry points for suspend/resume functionality.

16.5 Unit Test

The eSDHC driver is tested using the following tests included as part of the Windows CE 6.0 Test Kit (CETK).

- File System Driver Test
- Storage Device Block Driver Read/Write Test
- Storage Device Block Driver API Test
- Storage Device Block Driver Performance Test
- Partition Driver Test

16.5.1 Unit Test Hardware

Table 16-2 lists the required hardware to run the unit tests.

Table 16-2. Hardware Requirements

Requirement	Description
SD Cards	SanDisk (128MB, 512MB, Extreme III SDHC 4GB) ATP (SDHC 4GB) A-DATA Turbo (SDHC 4GB) Kingston (MiniSD 512MB, MicroSD 1GB)
MMC Cards	PQI (128 Mbytes) Kingmax (RS-MMC: 512MB, 1GB) Transcend (MMCPlus: 1 Gbytes, 4 Gbytes)

16.5.2 Unit Test Software

Table 16-3 lists the required software to run the unit tests.

Table 16-3. Software Requirements

Requirement	Description
tux.exe	Tux test harness, which is needed for executing the test
kato.dll	Kato logging engine, which is required for logging test data
tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
fsdtst.dll	File System Driver Test.dll file
rwtest.dll	Storage Device Block Driver Read/Write Test.dll file
disktest.dll	Storage Device Block Driver API Test.dll file
disktest_perf.dll	Storage Device Block Driver Performance Test
msparttest.dll	Partition Driver Test.dll file

16.5.3 Building the Unit Tests

All the above mentioned tests come pre-built as part of the CETK. No steps are required to build these tests. These test files can be found alongside the other required CETK files in the following location:

```
[Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I
```

16.5.4 Running the Unit Tests

The following are the tests available and the test procedures for each of the tests. For detailed information on the below tests see the relevant sub sections under **CETK Tests** in the Platform Builder Help, or view the online documentation at the following link: http://msdn2.microsoft.com/en-us/library/aa934353.aspx

16.5.4.1 File System Driver Test

The following command is used to run the tests on an SD card:

```
tux –o –d fsdtst –c “-p SDMemory –z”
```

For MMC cards, use the following command:

```
tux –o –d fsdtst –c “-p MMC –z”
```

Note that this tests all the cards inserted and requires the cards to be formatted prior to running the test. For higher capacity cards, the test takes a long time to complete, and hence it is recommended that the system power management (from control panel) be configured so that the system does not enter suspend state during test execution.

16.5.4.2 Storage Device Block Driver Read/Write Tests

The following command line is used to run the tests:

```
tux –o –d rwtest –c “-z”
```

NOTE

This command tests only one card at a time.

16.5.4.3 Storage Device Block Driver API Tests

The following command line is used to run the tests:

```
tux –o –d disktest –c “-z”
```

NOTE

This command tests only one card at a time.

16.5.4.4 Storage Device Block Driver Performance Tests

The following command line is used to run the tests:

```
tux –o –d disktest_perf –c “-z -disk DSK1:”
```

NOTE

This command tests only one card at a time.

16.5.4.5 Partition Driver Test

The following command line is used to run the tests:

```
tux –o –d msparttest –c “-z”
```

NOTE

The cards should be of size 256 Mbytes and higher. For higher capacity cards, the test takes long time to complete, and hence it is recommended that the system power management (from control panel) be configured so that the system does not enter suspend state during test execution.

16.5.5 System Testing

The following system tests are performed to verify the operation of the SD and MMC memory cards.

- Use the **Start** > **Settings** > **Control Panel** > **Storage Manager** to format and create partitions on the mounted memory cards.
- Establish ActiveSync connection over USB and transfer files to/from the memory cards.
- Write media files to memory storage. Use Windows Media Player to playback media files from memory storage.

16.6 Secure Digital Card Driver API Reference

Detailed reference information for the Secure Digital Card driver may be found in the Platform Builder Help under the heading *Secure Digital Card Driver Reference* or in the online documentation at the following link: http://msdn2.microsoft.com/en-us/library/aa912994.aspx

Chapter 17
Touch Panel Driver

The touch screen interface provides all the circuitry required for a 4-wire resistive touch screen. The touch screen X plate is connected to TSX1 and TSX2 and the Y plate is connected to TSY1 and TSY2. A local supply ADREF serves as reference.

17.1 Touch Panel Driver Summary

Table 17-1 provides a summary of source code location, library dependencies and other BSP information.

Table 17-1. Touch Panel Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\TOUCH
SOC Specific Path	
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\TOUCH
Driver DLL	touch.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX233 EVK:ARMV4I > Device Drivers > TOUCH > Touchscreen
SYSGEN Dependency	SYSGEN_TOUCH = 1
BSP Environment Variables	BSP_NOTOUCH= BSP_LRADC = 1

17.2 Supported Functionality

The touch panel should conform to the standards as explained in documentation at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Touch Screen Drivers

17.3 Hardware Operations

The hardware consists LRADC and touch screen interface. The touch screen controller configures the LRADC driver as required to do the measurement of the X and Y values of the touchscreen.

17.4 Software Operations

The touch screen driver reads user input from the touch screen hardware and converts the input to touch events. Then the touch screen events are sent to the Graphics, Windowing, and Events Subsystem (GWES). The driver also converts un-calibrated coordinates to calibrated coordinates. Calibrated coordinates compensate for any hardware anomalies, such as skew or nonlinear sequences.

For the touch screen driver to work properly it must submit points when the user finger or stylus is touching the touch screen. When the user finger or stylus is removed from the screen, the driver must submit at least one final event indicating that the user finger or stylus tip was removed. The calibrated coordinates must be reported to the nearest one-quarter of a pixel.

The following steps detail the basic algorithm that are used to sample and calibrate the screen with the touch screen driver:

1. Call the TouchPanelEnable function to start the screen sampling
2. Call the TouchPanelGetDeviceCaps function to request the number of sampling points

For every calibration point, perform the following steps:

1. Call TouchPanelGetDeviceCaps to get a calibration coordinate, a crosshair appears on the screen, touching the cross hair starts the calibration
2. Call the TouchPanelReadCalibrationPoint function to get calibration data
3. Call the TouchPanelSetCalibration function to calculate the calibration coefficients

17.4.1 Touch Driver Registry Settings

The following registry keys are required to load the touch driver:

```
IF BSP_NOTOUCH !
IF BSP_LRADC_TOUCH
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="touch.dll"
    "MaxCalError"=dword:7
IF BSP_PRECAL
    "CalibrationData"="539,520 280,259 280,778 793,781 794,259"

    ; Welcome.exe: Disable tutorial and calibration pages because we already
    ; have the necessary calibration data.
    ; Touch calibration (0x02), Stylus (0x04), Popup menu (0x08),
    ; Timezone (0x10), Complete (0x20)
[HKEY_LOCAL_MACHINE\Software\Microsoft\Welcome]
    "Disable"=dword:FFFFFFFF
ENDIF ; BSP_PRECAL

; For double-tap default setting
[HKEY_CURRENT_USER\ControlPanel\Pen]
    "DblTapDist"=dword:18
    "DblTapTime"=dword:637

; For launching the TouchPanel calibration application on boot.
[HKEY_LOCAL_MACHINE\init]
"Launch80"="touchc.exe"
"Depend80"=hex:14,00,1e,00 ; Wait for standard initialization
```

```
                                ; modules to load first (GWES.dll and
                                ; Device.exe).
ENDIF ; BSP_LRADC_TOUCH
ENDIF ; BSP_NOTOUCH !
```

17.5 Unit Tests

The following section explains about the hardware and software requirements for unit tests.

17.5.1 Unit Test Hardware

Table 17-2 lists the required hardware to run the unit tests.

Table 17-2. Hardware Requirements

Requirement	Description
LCD panel	Display panel required for display of graphics data

17.5.2 Unit Test Software

Table 17-3 lists the required software to run the unit tests.

Table 17-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Ktux.dll	Ktux.dll which is required to run in kernel mode
Touchtest.dll	The Test.dll File
Touch.dll	Touch Panel Driver

NOTE

The touch driver does work after the CETK Touch Panel Test. This is a known MSFT CETK issue. The MSFT online help notes that “When you complete the test, the operating system does not regain control of the touch panel. You must reset the touch panel to restore normal operation.” See the help topic at the following location: **CETK Tests and Test Tools** > **CETK Tests** > **Touch Panel Tests**

Cases 8011, 9001–9003 fail. The touch panel shows several lines when a circle or arc is drawn. This is also a known MSFT CETK issue. All these points are captured, but are not painted in time.

Case 8011 cannot draw in the right part of screen after a 90° rotation. ethca.exe works after rotation and the CETK works when the case runs again.

17.5.3 Running the Touch Panel Tests

The touch panel test cases can be run by typing:

```
tux -o -n -d touchtest.dll -x <Test case id>
```

The test case IDs are described in the documentation at the following location:

Windows Embedded CE Test Kit > CETK Tests and Test Tools >CETK Tests > Touch Panel Tests > Touch Panel Test

17.6 Touch Panel API Reference

The complete API reference is given in the documentation at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > Touch Screen Drivers > Touch Screen Driver Reference

Chapter 18
Universal Serial Bus (USB) On The Go (OTG) Driver

A USB OTG driver provides High Speed USB 2.0 host and peripheral support for the USB OTG port of the i.MX chip. The OTG driver automatically performs either a host or peripheral functionality at any given time, depending on the type of USB cable plugged in. There are three components to achieve this functionality: USB host driver, USB peripheral driver and USB OTG driver. The OTG driver maintains a state machine to decide whether a host driver or peripheral driver to be in charge. The OTG driver is also called as **Pin Detection Driver**, since the OTG functionality logic depends on the kind of USB cable plugged in.

Many class drivers are supported in WinCE. The host driver can be configured to work with mass storage, HID, printer, and RNDIS peripherals. The peripheral driver can be configured to provide mass storage, serial, or RNDIS functionality. The peripheral class supports are mutually exclusive, so that only one configuration can be selected as active configuration. The host functionality support do not have such limitation, and hence can recognize what kind of peripheral is plugged in and pick the right class driver to provide functionality.

Besides full OTG functionality, pure host driver and pure client driver options are also provided. These two options configure our BSP to work in either host-only or peripheral-only mode. In this case, pin detection driver is not active and no mode change will happen between the host and peripheral.

18.1 USB OTG Driver Summary

This section explains about the USB peripheral driver, host driver and OTG driver.

18.1.1 Peripheral Driver Summary

Table 18-1 lists the attributes of the peripheral driver.

Table 18-1. Peripheral Driver Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\SOC\<*Target SOC*>\USBD ..\SOC\<*Common Soc*>\ms\USBFN
CSP Static Library	usb_usbfn_<*Target SOC*>.lib usb_usbfn_os_<*Target SOC*>.lib usb_ufnmddbase_<*Common Soc*>.lib

Table 18-1. Peripheral Driver Summary (continued)

Driver Attribute	Definition
Platform Driver Path	\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBD
Import Library	NA
Driver DLL	usbfn.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed OTG Port Pure Client Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_CLIENT=1

USB peripheral class drivers are required to provide corresponding functionality. These class drivers are implemented as WinCE public driver. These class drivers (described in Section 18.4.7, "Peripheral Class Drivers") can be selected through drag and drop from catalog items.

18.1.2 Host Driver Summary

Table 18-2 lists the attributes of the host driver.

Table 18-2. Host Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX233-EVK
Target SOC (TGTSOC)	MX233_FSL_V2
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\SOC\<*Common SOC*>\ms\USBH\EHCI ..\SOC\<*Common SOC*>\ms\USBH\EHCIPDD ..\SOC\<*Common SOC*>\ms\USBH\USB2COM
CSP Static Library	usbh_ehcdmdd_<*Common SOC*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSOTG
Import Library	NA
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only host mode, choose .. > High Speed OTG Port Pure Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_HOST=1

USB host class drivers are required to provide corresponding functionality. As peripheral class drivers, the host class drivers are also implemented as WinCE public driver. See Section 18.4.8, "Host Class Drivers."

18.1.3 OTG (Pin-Detection) Driver Summary

Table 18-3 lists the attributes of the OTG driver.

Table 18-3. OTG Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX233-EVK
Target SOC (TGTSOC)	MX233_FSL_V2
CSP Driver Path	..\SOC\<Common Soc>\MS\USBOTG\MDD
CSP Static Library	usbotgcm_$(_COMMONSOCDIR)_PDK1_7.lib
Platform Driver Path	PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBOTG
Import Library	NA
Driver DLL	fsl_usbotg.dll
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed OTG Port Full OTG Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_CLIENT=1 BSP_USB_HSOTG_HOST=1 BSP_USBOTG=1

18.2 Supported Functionality

The OTG driver provides the following software and hardware support:

1. The High Speed OTG or Host driver supports USB specification 2.0
2. When no cable is connected or a mini-B cable is connected (in either of these cases, the ID pin is pull up), OTG driver select peripheral driver to be in charge. On attachment of a mini-A cable (in this case, the ID pin is pull down), OTG driver select the host driver to be in charge.
3. The peripheral driver can support mass storage, RNDIS, serial and basic personal healthcare classes. Only one class support is active.
4. The host driver can support mass storage, HID and Printer classes.
5. When nothing is attached to the OTG port, the driver configures the USB module to be in low power state.
6. When the system is suspended with nothing attached to the OTG or Host port, the system does not create a wake condition upon attachment of the port to a host or attachment of a device with mini-A plug.
7. When the system is suspended while the OTG or Host port is connected to a host or to a device with a mini-A plug, the system remains suspended when the OTG port connection is unplugged.
8. When the system resumes after suspend, any attached devices are enumerated and their class drivers loaded appropriately.

18.3 Hardware Operation

An EHCI compliant High-Speed OTG Controller and an USB 2.0 UTMI PHY are integrated on i.MX233 Chip. It provide a full on-chip USB OTG solution.

18.3.1 Conflicts with Other Peripherals and Catalog Items

This section explains USB OTG conflicts with other peripherals and catalog items.

18.3.1.1 Conflicts with SoC Peripherals

No conflicts.

18.3.1.2 Conflicts with Board Peripherals

The MX233 EVK board use pin D10 as both SSP1_DETECT (which is used in SDHC module) and USB_OTG_ID (which is used in USB OTG module host related functionalities). So Full OTG driver and pure host driver cannot coexist with SDHC driver. So in our default BSP configuration, full OTG is not selected. Instead, SDHC and pure Client Driver is selected.

To enable Full OTG functionality. Add **bsp_nossp1_sdhc = 1** in Environment Settings. Then select **High Speed OTG port Full OTG function** in catalog configuration.

18.4 Software Operation

This section explains about the software operation of the drivers.

18.4.1 USB OTG Host Controller Driver

This driver enables the USB host functionality for the OTG port. It is a part of the standard Windows USB software architecture.

Figure 18-1 shows the Windows USB driver architecture.

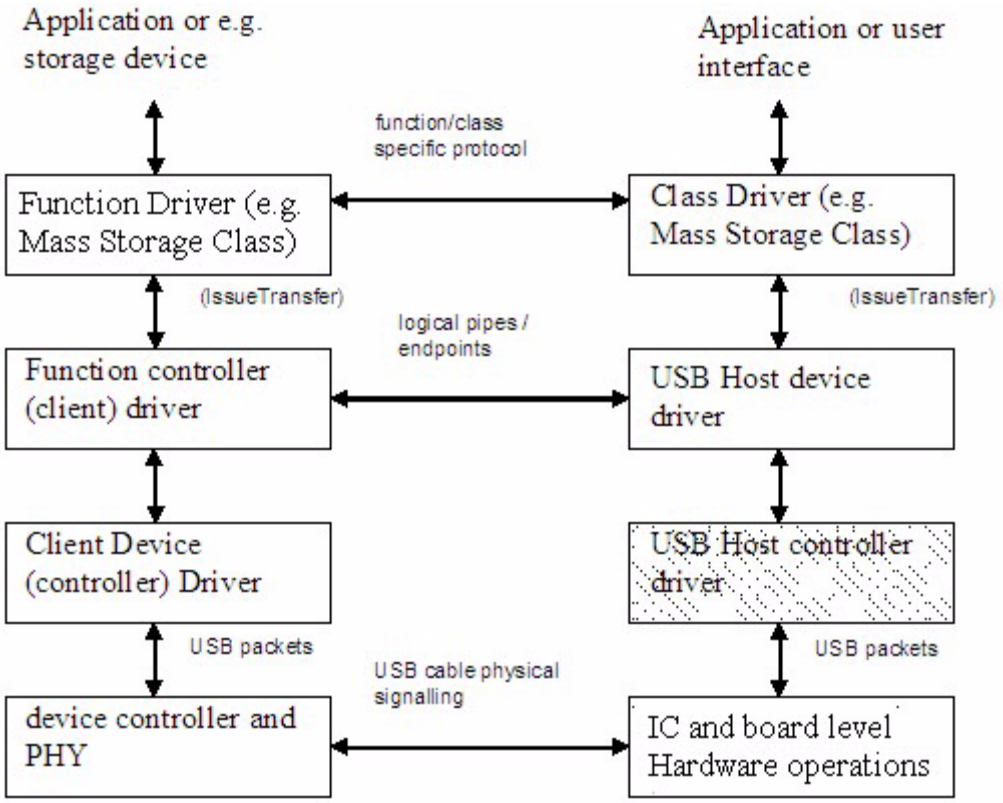

Figure 18-1. Windows USB Driver Architecture

The details about the Windows CE USB driver architecture and usage is found in the following location under the Platform Builder Help documentation at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers

and

Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers > USB Host Controller Drivers > USB Host Controller Driver Development Concepts

When the OTG driver is included, the host driver is activated when an USB Mini-A plug is connected to the Mini USB OTG socket. When pure host mode is selected, the host driver is always in control with respect to the relevant USB controller. When an USB peripheral device is connected, the host driver enumerates it and activates the appropriate class driver.

The BSP supports the following USB class drivers:

- Mass Storage—Card Reader with SD or CF cards, USB HDD drive, thumb drive (disk-on-key). Some card reader firmware is not supported by the Microsoft standard Mass Storage class driver
- HID—USB Keyboard and mouse
- RNDIS—Network Device Interface communication class

Hubs are also supported to extend the USB topology.

For more detailed description on host class driver, see the Section 18.4.8, “Host Class Drivers.”

18.4.1.1 User Interface

As described above, user can access to the USB host driver through class drivers. The details on the host client drivers is found under the Windows CE 6.0 Platform Builder Help documentation at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers > USB Host Controller Drivers > USB Host Client Drivers.

The new class driver code is to be developed using the documentation. See the host client driver interface functions (for example, IssueBulkTransfer) as documented in the Help topic:

Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers > USB Host Controller Drivers > USB Host Client Drivers > Host Client Driver Reference.

18.4.1.2 Host Controller Configuration

See Section 18.4.4, “USB OTG Catalog Settings,” for information about host controller configuration.

18.4.1.3 Memory Configuration

The USB Host drivers (for all USB host ports) use DMA to perform all USB transfers. The physical memory for these transfer buffers is allocated as a pool during driver initialization. Unless physical addresses are specified in API accesses at the class-driver interface, the driver copies data between the user or class-provided data buffers and the DMA buffer from the driver physical memory pool.

Host driver checks the registry key **PhysicalPageSize** for memory pool size. If it is not available or the registry setting is less than 128K, the driver uses the default minimal buffer size, 128K, and apply for the memory using **HalAllocateCommonBuffer**.

18.4.1.4 Configured Power

USB host driver monitors the configured power for all devices attached to a USB host. The host driver verifies that each attached device does not exceed the configured current limit.

This power limit is implemented through the platform-specific function `BSPUsbhCheckConfigPower()` as described in Section 18.4.1.8.1, “BSPUsbhCheckConfigPower,” and located in:

```
\PLATFORM\<Target Platform>\SRC\DRIVERS\USBH\Common\hwinit.c
```

This function is modified corresponding to the platform hardware capabilities. Currently we set the current limit to 500mA.

18.4.1.5 Registry Settings

See the Section 18.4.5, “USB OTG Registry Settings,” for the information about registry settings.

18.4.1.6 PHY level USB Test

The USB 2.0 specification defines PHY-level test modes for all the USB host ports (see section 7.1.20 for definitions in USB 2.0 specification). Temporarily this feature will not be enabled in our driver.

18.4.1.7 Unit Test

Different peripherals, such as thumb disk, keyboard and hub, are used to test the host driver functionality. Manual tests include connecting the peripheral, confirming the connection during plug in, during unplug and during subsequent plug in of device, data transfer verification (for mass storage peripherals) and other expected functionality such as keyboard, mouse and so on.

To verify the RNDIS class device, a CEPC containing Netchip 2280 USB function is attached and used to access a remote file server on the CEPC. To verify the low-level transport for bulk, interrupt and isochronous transfers, the CETK Host test kit is performed. This requires a CEPC configured with Netchip 2280 USB function and loopback driver.

18.4.1.7.1 USB Host Controller Driver Test

Documentation for the Windows CE 6.0 CETK USB Host tests is found under the Platform Builder Windows CE product documentation in the following location:

Debugging and Testing > **Windows CE Test Kit** > **CE Test Kit**

18.4.1.7.2 Build the Test Image

The following steps are used to build the test image:

1. Checkout the RTM to be tested or install the MSI provided
2. Add the following components from the catalog:
   - Freescale <Target Platform>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed Port Pure Host Function.
   - Core OS > Windows CE devices > Core OS Services > USB HOST Support; and all the sub-components of this catalog item (Sub-Components like USB Storage Class Driver.)
   - Core OS > Windows CE devices > File Systems And Data store > Storage Manager; (Sub-Components: FAT File System, Partition Driver, Storage Manager control panel applet)
   - Device Drivers > USB Function > USB Function Clients > Serial.

18.4.1.7.3 Abstract

This test suite can be used to test USB host controller drivers that provide the same interface as Window CE USB host controller driver does (for more information, see Section 18.4.1.1, “User Interface”), also it can be used to verify whether a certain USB host controller (either stand alone card or onboard logic) can operate with Windows CE.

Figure 18-2 shows the test setup and scenario.

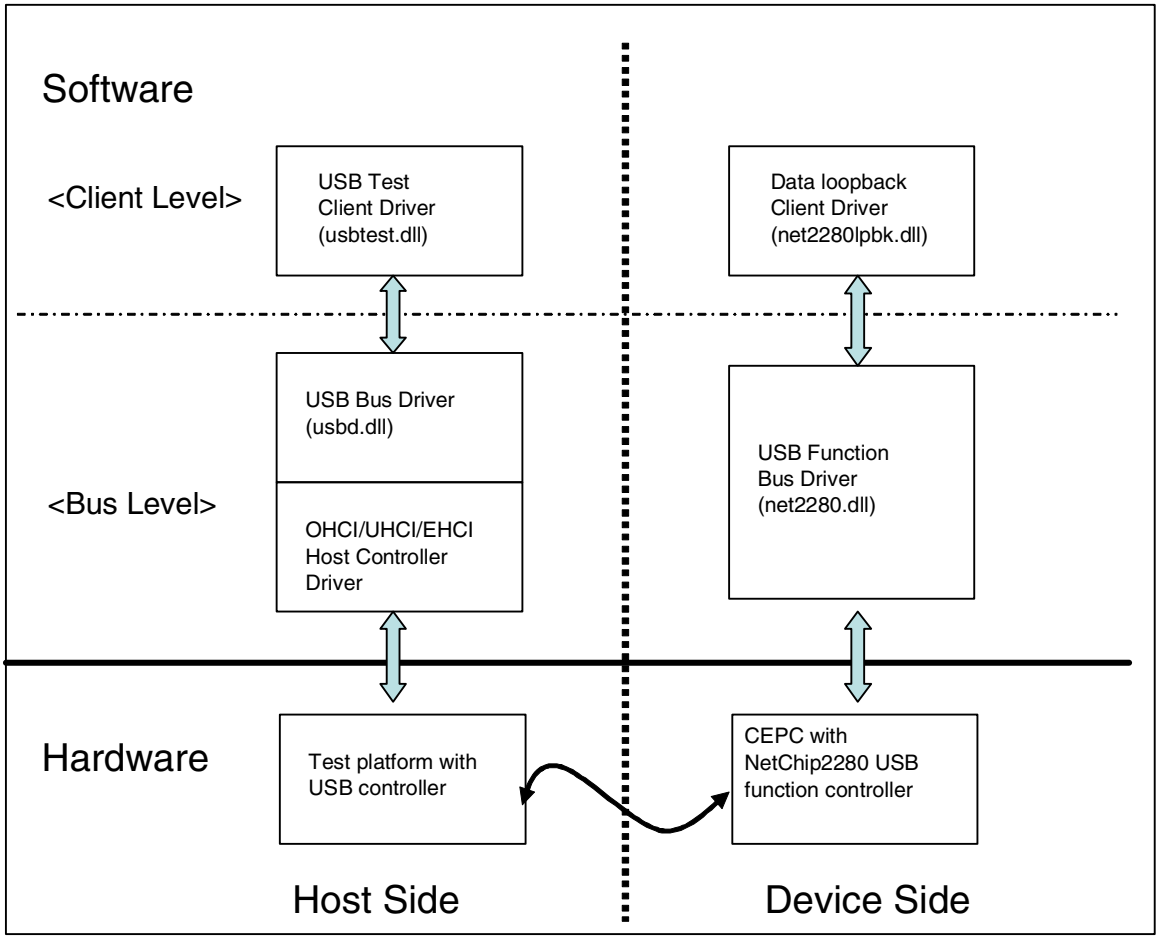

Figure 18-2. Test Setup

This test suite acts as a client driver above USB bus driver (usbd.dll). It is loaded when test device is connected to the host through the USB cable. The test device is a CEPC with a NetChip2280 USB function controller card in it. After this CEPC is booted up and net2280lpbk.dll is loaded, the CEPC acts as a generic USB data loopback device. USB test suite (the test client driver on the host side) can then stream data or issue device requests to or from this data loopback device. This is how the USB host controller and its corresponding host controller drivers are exercised.

NetChip2280 USB function PCI controller card is a USB2.0 compatible USB function device. It can be used to test both USB2.0 and USB1.1 host controllers (EHCI, OHCI or UHCI) and corresponding drivers.

Netchip2280 controller has six endpoints besides endpoint 0. The data loopback driver (net2280lpback.dll) configures these endpoints to be three pairs: one bulk IN or OUT pair, one Interrupt IN or OUT pair, and one Isochronous IN or OUT pair. The data loopback tests are done by sending data from host side to device side through OUT pipe, and receive it back through IN pipe, and then verify the data.

18.4.1.7.4 Unit Test Hardware

The following are the unit test hardware requirements:

- Test platform
- Host Controller Card (if not onboard logic)
- CEPC
- Netchip2280 Card
- USB cable

18.4.1.7.5 Unit Test Software

The following are the host side requirements:

- Tux.exe
- Ddlx.dll
- Usbtest.dll
- Tooltalk.dll
- Kato.dll
- USB component (usbd.dll, EHCI, OHCI or UHCI host controller driver(s)) must be included in the run time image.

The following are the device side requirements:

- Lufldrv.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image.

18.4.1.7.6 Running the Test

The test procedure is as follows:

1. Download runtime image to CEPC with Netchip2280 card on it
2. After system is boot up, run command `s lufldrv`, tester should verify that net2280lpbk.dll is loaded
3. Download runtime image to test platform with USB host controller on it
4. After system is boot up, make sure that there is no connection between host side and device through USB cable. Then launch command `s tux –o –d ddlx –c “usbtest” “–xYYYY”`, where `“YYYY”` is the test case(s) to be run
5. The test indicates that there should be no connection between host and device side. Then after seven seconds, the test asks to connect two sides with USB cable
6. The test main body starts to run
7. If there are other tests to be run, do not disconnect the USB cable. Type the next test command, and the tests starts directly. If the USB connection was disconnected before the next test, the test will ask to make the connection again.

18.4.1.7.7 Test Cases

Table 18-4 shows the test cases contain in the test suite.

Table 18-4. USB Host Controller Driver Test Cases

Test Case ID	Test Description
1001-1315, 1501-1515	Data loopback tests: Concerning the transfer type, there are five categories: 1) Bulk pipe loopback tests (tests with ID end with 1, like xxx1) 2) Interrupt pipe loopback tests (tests with ID end with 2, xxx2) 3) Isochronous pipe loopback tests (tests with ID end with 3, xxx3) 4) All pipe transfer simultaneously (tests with ID end with 4, xxx4) 5) All three types transfers carry on simultaneously (tests with ID end with 5, xxx5) [1] There are five categories for how data is transferred: 1) Normal loopback tests (tests with ID start with 10, like 10) 2) loopback tests using physical memory (tests with ID start with 11, 11xx) 3) loopback tests using a part of allocated physical memory (tests with ID start with 12, 12xx) 4) Normal short transfer loopback tests (tests with ID start with 13, 13xx) 5) Stress short transfer loopback tests (tests with ID start with 15, 15xx) Also both synchronous and asynchronous transfer methods are exercised (test cases like xx1x using asynchronous transfer method, test cases like xx0x using synchronous method There are a total of 5*5*2 = 50 test cases
1401-1413	Additional data loopback tests. that mainly focus on testing APIs like GetTransferStatus(), AbortTransfer() and CloseTransfer()
2001-2013	Test related to Device requests
9001-9004	Special tests that test APIs such as SuspendDevice(), ResumeDevice() and DisableDevice()
9005	Test that stresses EP0 transfer (Vendor Transfer)

[1] This category of tests is designed for testing some other USB function devices which have more endpoints than host controller driver can handle. When using Netchip2280, it should be the same as category 4). Tester can just ignore this category.

By default the data loopback device configures the endpoints with some often-used packet sizes that are DWORD aligned, and neither too big nor too small. By having all tests list above passed under this configuration is more than sufficient for a BVT-type test pass. However, testers can change the packet sizes (these values are hard-coded in the source code for net2280lpbk.dll) for each endpoint by themselves and run these test cases again for more comprehensive testing.

This test suite provides a way to change packet sizes of on NetChip2280 device on the fly. They are:

- Test case 3001: Using some very small packet sizes in NetChip2280 device full speed configuration
- Test case 3002: Using some very small packet sizes in NetChip2280 device high speed configuration
- Test case 3003: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device full speed configuration

- Test case 3004: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device high speed configuration
- Test case 3005 (High Speed only): Using some very large packet sizes (like 2*1024 for Isochronous endpoints) in NetChip2280 device full speed configuration. Note that in the real world, Netchip2280 cannot handle transfers using such large packet size because its onboard FIFO buffer is small.

What testers need to do is to run one of the test case above like running those normal tests, then after 15-20 seconds, automatically unload and load the usbtest.dll again through the Platform Builder. It means the packets sizes on Netchip2280 side have already been changed. Then those normal tests can be run. Use test case 3011 (for full speed config) and 3012 (for high speed) to restore the default packet sizes.

Another category test that is important for USB 2.0 host controllers and drivers is called the golden bridge tests, which means USB 2.0 host controller is connected with a full speed (USB 1.1) device. This is the only scenario that USB 2.0 host controller performs split transfers.

NetChip2280 can be forced to be a full speed device. In the test setup stage, instead of run `s lufldrv` to load loopback driver, run `s lufldrv -f`. This forces the Netchip2280 to be configured as a full speed device.

Also testers are encouraged to do some manual tests. Here are some examples:

- Plug in real USB devices, suspend system, and then resume; USB devices should still be there
- Plug in real USB devices, suspend system, unplug it, plug in another device, then resume; system should enumerate that new device properly
- Run one of the data transfer tests, in the middle of transfer stage, suspend the system (host side), then resume; tests may fail, but system should not crash
- Run one of the data transfer tests, in the middle of transfer stage, disconnect the USB connection; tests should fail, but system should not crash

18.4.1.8 Platform-Specific API

This section explains about the platform-specific APIs.

18.4.1.8.1 BSPUsbhCheckConfigPower

This function is used to evaluate whether a device can be supported on the specified USB port.

Parameters

UCHAR bPort	[in] Unused. Each USB controller has only one port
DWORD dwCfgPower	[in] Power requirement (number of milliamps) requested by the device being evaluated for attachment support on this port
DWORD dwTotalPower	[in] current total power (number of milliamps) used by other previously attached devices on this port
Return Value	Return TRUE if device requesting dwCfgPower can be safely attached Return FALSE if device can not be attached

18.4.1.8.2 BSPUsbSetWakeUp

This function does what is necessary to enable or disable wakeup on the USB port. This function does not actually enable wake-up when a device is currently attached to the port.

Parameters

BOOL bEnable [in] TRUE to enable wakeup, FALSE to disable wakeup

18.4.1.8.3 BSPUsbCheckWakeUp

This function evaluates the wake-up condition for the relevant USB port, and clears the condition and interrupt.

Parameters None

Return Value Return TRUE when a wake-up condition was detected
Return FALSE when no wake-up condition was present

18.4.1.8.4 SetPHYPowerMgmt

This function is called by the USB driver when transitioning to or from the suspended state (for example, during system suspend). The function does what is necessary to place the transceiver hardware into a suspended (fSuspend = TRUE) or running (fSuspend = FALSE) state.

The standard implementation for i.MX System uses a ULPI-bus based ISP1504 transceiver for the HS OTG port, and this function configures the ULPI-bus for sleep state. If platform hardware uses other transceivers, this function must be modified appropriately.

Parameters

BOOL fSuspend [in] TRUE: system or controller is going to suspend mode. FALSE: resuming

18.4.2 USB Peripheral Driver

This driver enables the USB peripheral functionality for the i.MX device. When this driver is active and the i.MX System is connected to a USB host system (for example, high speed or full speed port of PC), it is enumerated according to the current active configuration settings, and the appropriate class driver is loaded on the PC.

System can be configured as one of the following USB functions by setting the appropriate environment variable during build (drag or drop from the catalog).

- Serial class - Serial ActiveSync
- Mass storage class - expose local storage (ATA hard disk, RAMDISK or other store) as USB drive
- RNDIS class - Remote Network Driver Interface Specification
- PHD class - basic Personal Healthcare Device Class support

When multiple class supports are selected, only one class will be the active peripheral support. The default priority is: Serial Class > Mass Storage Class > RNDIS class > PHD class. Besides, we also provide tools to change current active class, see Section 18.7.1, “Application for USB Peripheral Class Driver Switch.”

For detailed description on peripheral class driver, see the Section 18.4.7, “Peripheral Class Drivers.”

18.4.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For more information on drivers, see the Help documentation at the following location:

Developing a Device Driver > Windows Embedded CE Drivers > USB Function Drivers > USB Function Controller Drivers.

User can access the USB client driver through function drivers such as Mass Storage or RNDIS. For further details on these USB Function drivers, see the following location in the Windows CE 6.0 Platform Builder help topic:

Developing a Device Driver > Windows Embedded CE Drivers > USB Function Client Drivers.

To get information where new function driver code is to be developed, see the Function controller driver interface functions (for example, IssueTransfer) as documented in:

Developing a Device Driver > Windows Embedded CE Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference.

18.4.2.2 Client Driver Configuration

See the Section 18.4.4, “USB OTG Catalog Settings” for information about the client driver configuration.

18.4.2.3 Registry Settings

See the 18.4.5, “USB OTG Registry Settings” for information about the registry settings.

18.4.2.4 PHY Test Mode

The USB 2.0 specification defines PHY-level test modes for USB device ports (see the section 7.1.20 for definitions in USB 2.0 specification). This mechanism allows a host to configure a device into test mode by commanding the device with a specific SET_FEATURE request. Once test mode is entered, the device is not able to leave test mode. Do not enable this feature in our BSP now.

18.4.2.5 Unit Test

There is no CETK test case for USB peripheral drivers. The USB Peripheral driver is tested manually for USB Serial function or USB Mass storage or RNDIS respectively. The test verifies basic USB peripheral functionality, including attach, detach, and data transfer. Separate images can be built and downloaded for each of the three peripheral function tests. See the Section 18.4.1.7.2, “Build the Test Image” for building the image. The peripheral class driver switch tool are also used to do these tests, see the Section 18.7.1, “Application for USB Peripheral Class Driver Switch.”

18.4.2.5.1 Unit Test Hardware

Table 18-5 lists the required hardware to run the unit tests.

Table 18-5. Hardware Requirements

Requirement	Description
Host system	a PC with proper driver and software installed
USB cable having Mini or Micro USB OTG plug A at one end and Mini or Micro USB OTG plug B on the other side	For connecting between the PC and peripheral
ATA, NAND, Thumb disk, SD Card or MMC card mounted on CE system	Required as a storage device when the board is configured as mass storage class

18.4.2.5.2 Unit Test Software

Table 18-6 shows the software requirements for the USB Function controller driver test.

Table 18-6. Software Requirements

Requirement	Description
ActiveSync 4.1 and above	Host side software that is required to be available for testing the Serial class functionality

18.4.2.5.3 Running the USB Function Controller Driver Tests

Table 18-7 lists USB Function controller driver tests:

Table 18-7. USB Function Controller Driver Tests

Test Cases	Entry Criteria, Procedure and Expected Results
Board configured as USB Serial class and connected to a host system after the board boots up completely	Entry Criteria: Make sure there is no cable connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini or micro USB OTG plug B to the mini or micro USB OTG socket 2. Observe that the ActiveSync on the host side gets connected and is synchronized 3. Copy files from Host system to the Mobile Device. Files are copied 4. Copy files from the Mobile Device to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the i.MX mini USB OTG socket to unload the Serial class driver Expected Result: ActiveSync should get synchronized and copying of files should happen between the Host and the System
Board configured as USB Mass storage client, with DSKx mounted, and connected to PC after the board boots up completely	Entry Criteria: Make sure there is no cable connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini or micro USB OTG plug B to the USB OTG socket 2. Observe that a new disk in My Computer having as Removable Disk appearing in it 3. Copy files from Host system to the new disk drive. Files are copied 4. Copy files from the new disk drive to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the mini USB OTG socket to unload the mass storage class driver Expected Result: Files copied into mass storage client device match those copied out (when compared on Windows XP PC using file compare utility). Note that files are not visible from within the System until the system has been reset. The file system should not be used inside the System when it is being accessed through USB as a mass storage client.
Board configured as USB RNDIS client and connected to a host system after the board boots up completely. Browsing the Internet	Entry Criteria: Make sure there is no cable connected and the board is turned on and wait until the board boots-up completely. See to it that the NIC's local area connection is not having any IP address Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that a new Local area connection in the Network and Dial up connections appears on the Windows XP machine. Bridge the NIC's local area connection and the RNDIS's local area connection 3. Configure the bridge by giving IP address, Subnetmask, Default gateway, DNS 4. On the System, a new Local area connection can be found in the Network and dial up connections. Configure the local area connection by giving IP address, Subnetmask, Default gateway, DNS 5. In the Internet explorer on the System, configure the Lan settings as per the local area settings Expected Result: Browsing the Internet should be possible

18.4.2.6 Platform-Specific API

This section explains about the functions that are platform-specific.

18.4.2.6.1 InitializeMux

This function is called to initialize the IOMUX connection within i.MX, from USB controller to the appropriate device pins for the transceiver. This function is implemented for the Pure Client situation.

Parameters

int Speed	[in] Unused
Return Value	Return TRUE if device requesting dwCfgPower can be safely attached

18.4.2.6.2 HardwarePullupDP

This function is called by the USB client driver when D+ must be pulled-up, in preparation for connection to a USB host. The standard code configures for ISP1504 or ISP1301 transceiver. It is possible to modify this routine to conditionally soft-disable USB connection.

Parameters

*CSP_USB_REGS *pRegs*	[in] pointer to the registers for the USB controller
Return Value	Return TRUE if D+ signal was pulled-up

18.4.3 USB OTG Driver (Pin-Detection Driver)

This driver is responsible for detecting the type of USB connector plugged into the USB OTG socket of the i.MX System. It loads the USB host driver or USB peripheral driver and let it in charge.

18.4.3.1 User Interface

There is no user interface to the transceiver driver. This driver merely manages the USB host or peripheral drivers, which provide the appropriate programming API.

18.4.3.2 OTG Driver Configuration

See the Section 18.4.4, “USB OTG Catalog Settings” for information on the OTG driver configuration.

18.4.3.3 Registry Settings

See the Section 18.4.5, “USB OTG Registry Settings” for information on the registry settings.

18.4.3.4 Unit Test

There is no CETK test case for USB OTG driver. It is tested using the mini or micro USB OTG plug A and mini or micro USB OTG plug B. The test is done by manually plugging in different cables to the OTG socket on the System and verifies if the appropriate driver is activated.

18.4.3.4.1 Unit Test Hardware

Table 18-8 lists the required hardware to run the unit tests.

Table 18-8. Hardware Requirements

Requirement	Description
Full OTG configuration selected in BSP	Make sure OTG driver is running
PC (with appropriate driver and software installed) Peripherals such as thumb disk, USB keyboard and hub	To test if control reaches the Host controller driver
mini or micro A to A receptacle cable mini or micro B to A cable	For connecting system with PC and peripherals. System acts as peripheral and host accordingly

18.4.3.4.2 Running the OTG Test

Table 18-9 lists OTG tests.

Table 18-9. OTG Tests

Test Cases	Entry Criteria, Procedure and Expected Results
Idle case when no cable plugged in	Entry Criteria: Make sure there is no cable connected and the board is turned on, wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, the board should be idle. Expected Result: Device boots up and is stable
Switch to peripheral	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, connect system to PC with the mini or micro B to A cable. Verify PC recognizes it correctly. Expected Result: PC recognize our board (as peripheral) correctly (Activesync be active, or removable disk be seen, or network adaptor be recognized).
Switch to host	Entry Criteria: Unplug board from PC (in previous step) Procedure: 1. Disconnect system with PC and connect a mini or micro A to A receptacle to the OTG socket. 2. Connect the USB peripheral device (such as a thumb disk) to the A receptacle. 3. The peripheral connected gets enumerated and starts functioning. For example, if a USB thumb disk is connected, a new disk will be accessible on CE system. Expected Result: Peripheral should start functioning on CE system
Switch between host and peripheral	Repeat the last 2 steps Expected Result: System always functions OK as both host and peripheral

18.4.3.5 Platform-Specific API

NA.

18.4.4 USB OTG Catalog Settings

The driver is selected into the BSP build by dragging and dropping the appropriate catalog item for USB HS OTG. There are three catalog items in **Freescale i.MX233 EVK: ARMV4I** > **Device Drivers** > **USB Devices** > **USB High Speed OTG** related to USBOTG functionality:

(a) **High Speed OTG Port Full OTG Function**

(b) **High Speed OTG Port Pure Client Function**

(c) **High Speed OTG Port Pure Host Function**

The selection of (a) will implicitly select (b) and (c), without selecting (a), (b) and (c) separately. So there are three possible configurations available for BSP users:

(1) All 3 catalogs are explicitly or implicitly selected, corresponding to both host and peripheral support plus OTG pin detection.

(2) Only **High Speed OTG Port Pure Client Function** is selected, corresponding to peripheral-only support.

(3) Only **High Speed OTG Port Pure Host Function** is selected, corresponding to host-only support.

18.4.5 USB OTG Registry Settings

The three possible configurations presented in Section 18.4.4, "USB OTG Catalog Settings," make three corresponding registry structure.

18.4.5.1 Registry Structure

- With configuration 1, for full OTG configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg]
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg\USBFN]
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg\Hcd]
```

- With configuration 2, for full peripheral-only configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UFN]
```

- With configuration 3, for full host-only configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSOTG]
```

The contents in **BuiltIn\USBOtg\UsbFN** are quite similar to those in **BuiltIn\UFN** and the contents in **BuiltIn\UsbOtg\Hcd** are quite similar to those in **BuiltIn\HCD_HSOTG**. They share most of the settings. The difference is that:

In configuration 1, only **UsbOtg** key is located under **BuiltIn** key, which means the OTG driver will automatically loaded by the OS. In this case, the OTG driver will decide to load the peripheral driver and the host driver.

In configuration 2 and 3, **UFN** or **HCD_HSOTG** is put directly under **BuiltIn** key. So the peripheral driver or host driver will be loaded by the OS automatically.

18.4.5.2 Registry Key Settings

This section explains about the registry key settings.

18.4.5.2.1 OTG Driver Settings

Table 18-10 lists the USB OTG transceiver registry settings.

Table 18-10. USB OTG Transceiver Registry Settings

Value	Type	Content	Description
Dll	sz	fsl_usbotg.dll	Driver dynamic link library
IsrDll	sz	giisr.dll	ISR Chain Handler
DynamicClientLoad	dword	3	We set the value to 0x3, indicating both host driver and peripheral driver will be loaded dynamically by OTG driver

18.4.5.2.2 Peripheral Driver Settings

Table 18-11 lists the USB OTG client registry settings.

Table 18-11. USB OTG Client Registry Settings

Value	Type	Content	Description
Dll	sz	usbfn.dll	Driver dynamic link library
OTGSupport	dword	0	obsolete setting, must be set as 0
Priority256	dword	64	The reference peripheral driver IST priority
OTGGroup	sz	1	This unique string (example 00 to 99) is used to combine or correlate instances of the host, function, and transceiver driver within one USB OTG instance

18.4.5.2.3 Host Driver Settings

Table 18-12 lists the default values for the host driver settings.

Table 18-12. hsotg.reg Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsotg.dll	Driver dynamic link library
OTGSupport	dword	0	obsolete setting, must be set as 0
OTGGroup	sz	01	This unique string (example “00” to “99”) is used to combine or correlate instances of the host, function, and transceiver driver within one USB OTG instance.

Table 18-12. hsotg.reg Default Values (continued)

Value	Type	Content	Description
HcdCapability	dword	4	HCD_SUSPEND_ON_REQUEST. Note: HCD_SUSPEND_RESUME is always assumed.
PhysicalPageSize	dword	NA	This value represents the number of bytes allocated for the physical memory pool of the OTG host driver, and defaults to 128kB. From this buffer, 75% are allocated for transfer descriptors and the remaining buffer is available for allocation to simultaneous transfers. In most cases, only one transfer is active at any time (for example, in the Mass Storage Class). A good value is at least 3x as large as the largest data buffer transferred using IssueTransfer(). Our BSP don't provide this setting and the driver will use the default 128kB size

18.4.6 Power Management

The USB OTG driver enters the low power mode in the following cases:

- No bus activity for a specified period of time
- System enter suspend state

Similar procedures are taken to let the USB module to enter or exit low power mode in either of the 2 cases. The following section explains about the description on the general power management procedures.

18.4.6.1 Power Down Procedure

To set the USB module to low power mode, both PHY and controller should be set to low power mode respectively.

18.4.6.1.1 Set PHY to Low Power Mode

The following function is called to set the PHY to low power mode:

```
BSPUsbPhyEnterLowPowerMode(pUsbRegs, TRUE)
```

This function is defined in

`..\platform\<Target Platform>\src\drivers\usbcommon\usbutils.c`

The following procedure is used for setting the PHY to low power mode.

- enable the wakeup interrupt source which can be activated without USB clock
- close power to all PHY sub module

18.4.6.1.2 Close USB Controller Clock

The following function is called to close the USB controller clock:

```
BSPUSBClockSwitch(FALSE)
```

This function is defined in

`..\platform\<Target Platform>\src\drivers\usbcommon\usbclock.c`

It gate the IC clock to USB Controller module.

18.4.6.2 Power Up Procedure

The USB module is powered up by reversing the procedures that are used to exit the low power mode.

18.4.6.2.1 Open USB Controller Clock

The following function is called to open the USB controller clock:

```
BSPUSBClockSwitch(TRUE)
```

It ungate the IC clock to USB Controller module.

18.4.6.2.2 Put PHY Out of Low Power Mode

The following function is called to retrieve the PHY from low power mode:

```
BSPUsbPhyEnterLowPowerMode(pUsbRegs, FALSE)
```

The following procedures are implemented to set the PHY out of low power mode.

- disable the wakeup interrupt source which can be activated without USB clock
- open power to all PHY sub module

18.4.6.3 Processing Methodology

This section explains how to integrate the power down and power up procedure into USB OTG driver. Since the USB OTG driver include the OTG driver, the host driver and the peripheral driver, the processing methodology for all the three drivers is discussed in this section.

18.4.6.3.1 Host Driver Methodology

1) Auto low power

The host driver IST wait for USB IRQ for a specified interval of time. The interval is defined as a macro **USB_IDLE_TIMEOUT**, which is set to 3000ms in the BSP. If there are no USB IRQ during this period, there is nothing to be connected, so the driver will follow the procedure as described in Section 18.4.6.1, “Power Down Procedure,” to set the USB module to low power mode.

When the module is in low power mode, the driver is sensitive to the USB interrupt. Once such an interrupt is caught, the driver will follow the procedure described in Section 18.4.6.2, “Power Up Procedure,” to set the USB module out of low power mode and function normally.

The implementation is found in

`CHW::UsbInterruptThread`, which is located in

`SOC\<Common SOC>\ms\USBH\EHCI\chw.cpp`

2) Low power mode with system suspend

When the system enters suspend mode, the USB module will enter the low power mode. The power down procedures as described in Section 18.4.6.1, “Power Down Procedure,” are also implemented in host driver

`CHW::PowerMgmtCallback`, which is located in

SOC\<*Common SOC*>\ms\USBH\EHCI\chw.cpp

This function will be called by OS automatically during system suspend.

When the system exits suspend mode, the USB module also exit the low power mode. The power up procedures as described in Section 18.4.6.2, "Power Up Procedure," are also implemented in

CHW::PowerMgmtCallback

This function will be called by OS automatically during system resume.

18.4.6.3.2 Peripheral Driver Methodology

1) Auto low power

The peripheral driver IST wait for USB IRQ for a specified interval of time. The interval is defined as a macro **IDLE_TIMEOUT**, which is set to 3000ms in our BSP. If there are no USB IRQ during this period, there is nothing to be connected, so the driver will follow the procedure as described in Section 18.4.6.1, "Power Down Procedure," to set the USB module to low power mode.

When the module is in low power mode, the driver is sensitive to USB interrupt. Once such an interrupt is caught, the driver will follow the procedure as described in Section 18.4.6.2, "Power Up Procedure," to set the USB module out of low power mode and function normally.

The implementation can be found in

InterruptHandle, which is located in

SOC\<*Common SOC*>\ms\USBD\COMMON\pdd.c

2) Low power mode with system suspend

When the system enters suspend mode, the USB module also enters the low power mode. The power down procedures as described in Section 18.4.6.1, "Power Down Procedure," are implemented in peripheral driver in

UfnPdd_PowerDown

Which is located in

SOC\<*Common SOC*>\ms\USBD\COMMON\pdd.c

This function will be called by OS automatically during system suspend.

When the system exits suspend mode, the USB module also exit low power mode. The power up procedures as described in Section 18.4.6.2, "Power Up Procedure," are implemented in

UfnPdd_PowerUp

Which is also located in

SOC\<*Common SOC*>\ms\USBD\COMMON\pdd.c

18.4.6.3.3 OTG Driver Methodology

At any time after system is boot up, either host driver or peripheral driver is in charge of the USB module. When USB module need to enter or exit low power mode, all the tasks are done by the in-charge driver. So there is no need for OTG Driver to provide redundant processing.

18.4.6.4 USB Wakeup

For some system design, it is preferred for the host driver that after system goes into suspend mode, an USB action by peripheral (such as plug, unplug and so on.) can wakeup the whole system. It is not implemented in i.MX233 yet.

18.4.7 Peripheral Class Drivers

The function drivers can be configured using the Windows CE 6.0 Platform Builder catalog, and are located at:

Device Drivers > USB Function > USB Function Clients

Besides that, basic Personal Health Care Class (PHCC) support is also included in the BSP, the catalog is

Third Party > BSP > Freescale <Target Platform>: ARMV4I > Device Drivers > USB Devices > USB Functional Class Driver > Personal HealthCare Class Support

The default function driver is launched, when the USB device port is attached to a host. This default function driver is selected by the registry key (the last instance of this value in reginit.ini applies):

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
   "DefaultClientDriver"=- ; erase previous default
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
   "DefaultClientDriver"="Mass_Storage_Class"
```

or

```
   "DefaultClientDriver"="RNDIS"
```

or

```
   "DefaultClientDriver"="Serial_Class"
```

or

```
   "DefaultClientDriver"="Personal_HealthCare_Class"
```

18.4.7.1 Mass Storage Function

Table 18-13 lists the mass storage functions.

Table 18-13. Mass Storage Function

Driver Attribute	Definition
CSP Driver Path	..\SOC\<*Common SOC*>\ms\USBFN\CLASS
CSP Static Library	NA
Platform Driver Path	NA
Import Library	USBMSFN_LIB_<*Common SOC*>.lib UFNCLIENTLIB.LIB
Driver DLL	usbmsfn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Mass Storage
SYSGEN Dependency	SYSGEN_USBFN_STORAGE

The Mass Storage function exposes a local data store as a USB peripheral storage device. The device used can be specified in registry. In platfrom.reg, the following template is provided:

```
PUBLIC\Common\OAK\Files\common.reg
"DeviceName"=- ;
; "DeviceName"="ATA HARD DISK"
; "DeviceName"="SDMEMORY CARD"
; "DeviceName"="MMC CARD"
; "DeviceName"="USB HARD DISK"
; "DeviceName"="NAND FLASH"
```

Any item from this list can be specified to act as the mass storage medium. Uncomment the corresponding line and rebuild the BSP to make that item active.

If none of the items are specified explicitly, a pre-coded priority is used to determine what active drive acts as a mass storage medium. The priority is described as the following:

```
ATA HARD DISK > SDMEMORY CARD (MMC CARD) > USB HARD DISK > NAND FLASH
```

The platform.reg can also over-ride other USBMSFN related default settings. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Mass_Storage_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:FFFF
   "Product"="Generic Mass Storage (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

18.4.7.2 Serial Function

The primary use for serial function is ActiveSync. Table 18-14 lists the serial functions.

Table 18-14. Serial Function

Driver Attribute	Definition
CSP Driver Path	NA
PUBLIC driver path	PUBLIC\Common\OAK\Drivers\USBFN\CLASS\SERIAL
CSP Static Library	NA
Platform Driver Path	NA
Export Library	serialusbfn.lib
Import Library	com_mdd2.lib serpddcm.lib ufnclientlib.lib
Driver DLL	SerialUsbFn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Serial Client
SYSGEN Dependency	SYSGEN_USBFN_SERIAL

NOTE

ActiveSync has been tested using the connection to PC with the ActiveSync version 4.1 or above installed. See www.microsoft.com to download the latest ActiveSync software for the PC. In some cases, DEBUGCHK may be triggered during attachment to ActiveSync in DEBUG builds.

When SYSGEN_USBFN_SERIAL is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:00ce
   "Product"="Generic Serial (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

18.4.7.3 RNDIS Function

The RNDIS function allows communication over USB to be supplied to ethernet NDIS interface of protocol stack. Table 18-15 lists the RNDIS functions.

Table 18-15. RNDIS Function

Driver Attribute	Definition
CSP Driver Path	NA
CSP Static Library	NA
Platform Driver Path	NA
PUBLIC Driver Path	PUBLIC*\OAK\Drivers\USBFN\Class\RNDIS
Import Library	ndis.lib
Driver DLL	RNDISFN.DLL
Catalog Item	Device Drivers > USB Function > USB Function Clients > RNDIS Client
SYSGEN Dependency	SYSGEN_USBFN_ETHERNET

RNDIS function has been tested using the Freescale RNDIS class driver located at:

```
Support\RNDIS\ce6_rndis.inf
%WINDIR%\System32\drivers\usb8023x.sys
```

When SYSGEN_USBFN_ETHERNET is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\RNDIS]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"idVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"idProduct"=dword:0301
"Product"="Generic RNDIS (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

18.4.7.4 PHDC Function

PHDC collects the personal health related data such as glucose meters and temperature measurements from portable devices, and then transmit the data to the center agent, for example, PC or health care center host. Table 18-16 lists the PHDC functions.

Table 18-16. PHDC Function

Driver Attribute	Definition
CSP Driver Path	..\SOC\<*Common SOC*>\UFNCLASS\CLASS\PHDC
PUBLIC driver path	NA
CSP Static Library	NA
Platform Driver Path	NA
Export Library	NA
Import Library	NA
Driver DLL	usbphdfn.dll
Catalog Item	Third Party > BSP > Freescale <Target Platform>: ARMV4I > Device Drivers > USB Devices > USB Functional Class Driver > Personal HealthCare Class Support
SYSGEN Dependency	NA
BSP Variable	BSP_USBFN_PHD_SUPPORT

As it is a non Microsoft provided class driver, the PHDC class driver currently support basic reliable personal health data transfer in the continua alliance framework. An peripheral side API is developed to transfer the multiple personal health measurement, including weight, glucose, blood pressure and temperature to a PC installed with proper PHDC host driver and application. For more details, see Section 18.7.2, “Application for Multispec PHDC Demo.”

18.4.8 Host Class Drivers

All host ports support the same class drivers, and this configuration is common to all host ports. Class drivers must also be configured for the USB host ports. Class driver configuration is common to all host

ports; there is no port-specific configuration for on any class driver. Table 18-17 shows the standard Microsoft-supplied drivers, and these drivers can be dragged and dropped from the catalog.

Table 18-17. Class Drivers

Class Driver	Configuration Flag	Catalog Item
HID	SYSGEN_USB_HID	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Human Input Device (HID) Class Driver
Printer	SYSGEN_USB_PRINTER	.. > USB Printer Class Driver (and see additional configuration in Section 18.6.2, "Dependencies of Drivers")
Keyboard	SYSGEN_USB_HID _KEYBOARD	.. > Keyboard HID Device (and see additional configuration in Section 18.6.2, "Dependencies of Drivers")
Mouse	SYSGEN_USB_HID _MOUSE	.. > Mouse HID Device (and see additional configuration in Section 18.6.2, "Dependencies of Drivers")
RNDIS	SYSGEN_ETH_USB_HOST	.. > USB Remote NDIS Class Driver
Storage	SYSGEN_USB_STORAGE	.. > USB (mass) Storage Class Driver

18.4.8.1 HID Mouse

For mouse support, the cursor is required to test or use the mouse. Table 18-18 shows the HID mouse class drivers.

Table 18-18. HID Mouse Class Driver

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS > Shell and User Interface > User Interface > Customizable UI > Mouse

18.4.8.2 HID Keyboard

The System keyboard key mapping conflicts with the HID keyboard. So, when the USB keyboard is included, remove the System keyboard and include the appropriate stub keyboard and keyboard.dll file. Table 18-19 lists the HID keyboard driver that is to be removed.

Table 18-19. HID Keyboard Driver to Remove

Remove Item	Remove Catalog Item
Keyboard	Third Party > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Input Devices > Keyboard US or Keypad

Table 18-20 lists the stub keyboard driver that is to be included.

Table 18-20. ID Keyboard Driver to Include

Catalog Item	Configuration Flag	Catalog Item
NOP Stub Keyboard	BSP_KEYBD_NOP	Device Drivers > Input Devices > Keyboard or Mouse > NOP (Stub) Keyboard or Mouse English

And include the appropriate keyboard.dll. For example, define SYSGEN_KBD_US and add the following lines in the platform.bib (immediately before the FILES section):

```
IF BSP_KEYBD_NOP
    kbdmouse.dll    $(_FLATRELEASEDIR)\KbdnopUs.dll                    NK SH
ENDIF; BSP_KEYBD_NOP
```

18.5 Known Issues

This section provides known issues of current BSP. These issues are mainly caused by hardware limitation, and hence no software work-around is available.

18.5.1 Host Support for Low Speed Peripherals

The i.MX233 USB PHY do not support low speed device. So a large number of USB mouse or keyboard cannot be recognized if they are connected directly to the root hub. So to utilize them, a high-speed hub must be used to be the bridge.

Connecting a low speed device through a full speed hub also do not make sense.

18.5.2 Host VBUS Power Supply

The i.MX233 EVK board design do not provide software to drive VBUS to 5V, that makes host unable to detect the peripheral connection.

A hardware work-around is to add a jumper on Q8 pin 2 and pin 3. Since pin 2 is lined to 5V wall supply and pin 3 is lined to VBUS line, shorting both the pins will provide a necessary voltage to VBUS and get host working.

18.6 Basic Elements for Driver Development

This section provides details of the basic elements for driver development in the Platform System.

18.6.1 BSP Environment Variables

Table 18-21 summarizes the System environment variables.

Table 18-21. System Environment Variables Summary

Name	Definition
BSP_USBOTG	Set to enable Full OTG functionality (enable host-client switching) on the High Speed OTG port
BSP_USB_HSOTG_CLIENT	Set to include USB client functionality on High Speed OTG port
BSP_USB_HSOTG_HOST	Set to include USB host functionality on High Speed OTG port.

18.6.2 Dependencies of Drivers

Table 18-22 summarizes the Microsoft-defined environment variables used in the BSP.

Table 18-22. USB Driver

Name	Definition
SYSGEN_USBFN_SERIAL	Set to support serial class for USB Function controller
SYSGEN_USBFN_STORAGE	Set to support mass storage class for USB Function controller
SYSGEN_USBFN_ETHERNET	Set to support RNDIS class for USB Function controller
SYSGEN_CURSOR	Set to support mouse cursor
SYSGEN_FATFS	Set to support FAT16 file system
SYSGEN_PCL	Set to support PCL printing
SYSGEN_PRINTING	Set to support printer
SYSGEN_STOREMGR	Set to support storage manager
SYSGEN_UDFS	Set to support Universal Disc File System
SYSGEN_USB	Set to support USB driver
SYSGEN_USB_HID	Set to support Human Interface driver (HID) class
SYSGEN_USB_HID_CLIENTS	Set to support HID clients
SYSGEN_USB_HID_KEYBOARD	Set to support HID keyboards (keyboard stub and associated.dll are required)
SYSGEN_USB_HID_MOUSE	Set to support HID mouse
SYSGEN_USB_PRINTER	Set to support Printer (printer driver support, such as PCL (SYSGEN_PCL), may be required)
SYSGEN_USB_STORAGE	Set to support storage medium

18.7 Application Tools for USB

This section describes about the application tools that are used for the USB.

18.7.1 Application for USB Peripheral Class Driver Switch

Only one USB peripheral drivers can be active even if there are many. When multiple class drivers are included in the image, it will be convenient that it can be switched. It is convenient for both the end users and test engineers. The following executable programs are added in ..\platform*<Target Platform>*\files:

switchUsb2Msc.exe, switchUsb2Rndis.exe, switchUsb2Serial.exe, switchUsb2Phdc.exe.

These executable programs are selectively integrated into nk.bin during the generation of OS image. During WinCE start up, the programs can be found in \WINDOWS directory. On execution of these programs activate the mass storage, RNDIS, Serial or PHDC peripheral drivers respectively.

18.7.2 Application for Multispec PHDC Demo

A peripheral side GUI is needed to use the PHDC class driver. The application is used to select any data from the available personal health measurements and send the selected data to PC. The PHDC_Peripheral_App.exe is located in the directory ..\platform*<Target Platform>*\files.

During WinCE start up, the file is found in \WINDOWS directory.

On the host side, Continua Alliance CESL Reference Software is necessary to setup PHDC communication channel. Contact www.continuaalliance.org for more details.

When PC side is ready, run the PHDC_Peripheral_App.exe. It will brings up a GUI with the following button controls

- **Select Device Spec** Button—Used to select between the four different personal health data category.
- **Send Measurement** Button—Used to send current measurement data category to host.
- **Disconnect** Button—Used to test PHDC protocol level disconnect.

Chapter 19
USB Boot and KITL

USB Boot and KITL are supported by implementing a RNDIS client device over USB on the target board. This feature configures the USB OTG port as a USB device and implements the RNDIS USB function driver. The USB RNDIS device acts as a normal ethernet device and connects to the PC over a USB cable. Eboot and KITL then operate with the RNDIS ethernet device.

19.1 USB Boot and KITL Summary

Table 19-1 identifies the source code location, library dependencies, and other BSP information.

Table 19-1. USB Boot and KITL Summary

Driver Attribute	Definition
Target Platform	iMX233-EVK
Target SOC	MX233_FSL_V2
SOC Common Path	WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\MS\RNE_MDD WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\MS\USBKITL
SOC Specific Path	WINCE600\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBD\KITL
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\COMMON\USBFN ..\PLATFORM\<*Target Platform*>\SRC\KITL
Driver DLL	fsl_usbfn_rndiskitl.lib
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variable	N/A

19.2 Supported Functionality

The USB Boot and KITL provides the following software and hardware support:

1. Image downloading over USB RNDIS
2. KITL over USB
3. Provides menu options to determine whether or not to enable USB Boot and/or USB KITL

19.3 Hardware Operation

For detailed operation and programming information of the USB OTG, see the chapter on the High-Speed USBOTG_UTMI in the corresponding platform User's Guide.

19.3.1 Conflicts with Other Peripherals and Catalog Items

The USB Boot and KITL does not have conflicts with any other module. However, since USB KITL and USB OTG drivers share the same USB OTG hardware, the USB OTG drivers should be disabled in the catalog item when USB KITL is enabled. USB boot does not have such limitation.

19.4 Software Operation

This section explains about the software requirements for USB OTG.

19.4.1 Software Architecture

USB Boot and KITL are part of the EBOOT and KITL subsystem. A RNDIS client device is implemented to support USB Boot and KITL. Figure 19-1 illustrates the USB Boot and KITL software architecture.

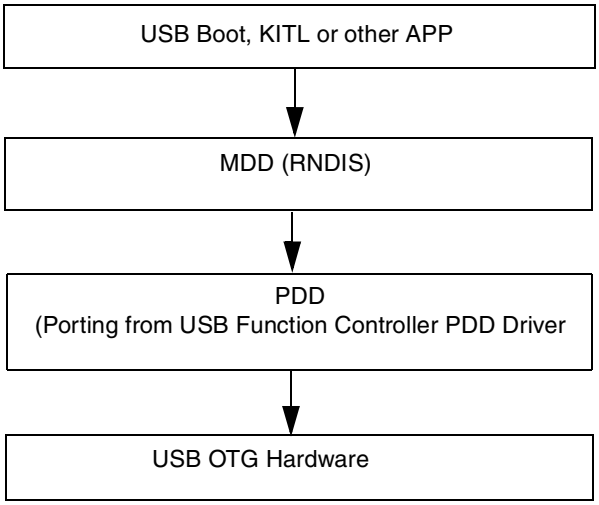

Figure 19-1. USB Boot and KITL Software Architecture Block Diagram

Microsoft has implemented a RNDIS client MDD driver in Windows CE 6.0. The code is in following location:

```
%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg\Rne_mdd
```

It generates the static library `Rne_mdd.lib`.

The USB function controller PDD driver is ported to eboot and KITL to support USB Boot and KITL. For details of USB function controller PDD driver see the Platform Builder Help in the following location:

Developing a Device Driver > Windows Embedded CE Drivers > USB Function Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference > USB Function Controller PDD Functions.

Windows CE 6.0 provides an example of USB Boot. It is located at:

```
%_WINCEROOT%\Platform\MainstoneIII\Src\Common\Usbfn
```

19.4.2 Source Code Layout

Some files are modified or added to support USB Boot and KITL. They are as follows:

- RNDIS PDD driver
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\COMMON_FSL_V2\MS\USBKITL\RNDIS`
- USB function controller shared with OS driver
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\COMMON`
- Add RNDIS device to EBOOT ethernet initialization routines
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\ether.c`
- Setup KITL device LogicalLoc and PhysicalLoc to USBOTG physical address if USB KITL option in EBOOT menu is selected by user
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\main.c`
- Implement private OS functions, such as `NKCreateStaticMapping(). NKCreateStaticMapping` is defined in OS. It is not defined for EBOOT while USB Boot requires this function. So it is manually defined. This function just calls OALPAtoUA()
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\KITL`
- Add USB Boot and KITL options into EBOOT menu
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\menu.c`
- Add `fsl_rne_mdd_$(_COMMONSOCDIR).lib, fsl_rne_pdd_$(_COMMONSOCDIR).lib, usb_usbfn_$(_SOCDIR).lib, usb_usbfn_eboot_$(_SOCDIR).lib`
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\sources`
- Add USB RNDIS KITL device in KITL initialization routines
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\kitl.c`
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\sources`

19.4.3 Power Management

Power management is not implemented in USB Boot and KITL.

19.4.4 Registry Settings

There are no related register settings for the USB Boot and KITL.

19.4.5 DMA Support

Physical contiguous memory is required to support USB DMA. This memory region is hard coded in:
`%_WINCEROOT%\Platform\Common\SRC\SOC\<Common Soc>\ms\Usbkitl\Rndis\rndis_pdd.c`

It uses the BSP reserved IPL RAM image region (Starting from IMAGE_USB_KITL_RAM_PA_START). This region is not used by other modules in the BSP, so it can be used by USB boot and KITL.

19.5 Unit Test

The following section explains how to perform unit tests.

19.5.1 Building the USB Boot and KITL

There is no special configuration options for building USB Boot and USB KITL. Building the BSP with default configuration includes the USB Boot and KITL support. The exception is that the USB OTG drivers should be deselected from the catalog item view before building the NK image to use USB KITL, because USB KITL and OS USB drivers share the same USB OTG hardware and they can not exist simultaneously. As a result USB KITL can not used to debug USB OTG drivers.

The USB OTG driver auto unloads when it detects USB KITL enabled.