Windows CE BSP

i.MX31ADS, i.MX32ADS Reference Guide

Version WCE500-14
October 18, 2007

06-4752-RM-TX30

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

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)
support@freescale.com

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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

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

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

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

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

Contents

About This Book

Audience xxi
Conventions xxi
Definitions, Acronyms, and Abbreviations xxi
Suggested Reading xxii
References xxiii

Chapter 1 Introduction

1.1 Overview 1-1
1.2 BSP Scope 1-1
1.3 Features 1-1
1.4 Getting Started 1-1
1.5 Windows CE Architecture 1-2

Chapter 2 ATA Driver

2.1 ATA Driver Summary 2-1
2.1.1 MX31 and MX32 2-1
2.2 Requirements 2-1
2.2.1 MX31 and MX32 2-1
2.3 Hardware Operation 2-2
2.3.1 ATA Hardware Block Diagram 2-3
2.3.2 Conflicts with other Peripherals and Catalog Options 2-3
2.3.3 Cabling 2-4
2.4 Software Operation 2-4
2.4.1 Application / User Interface to ATA drives 2-4
2.4.2 ATA Driver Configuration 2-4
2.4.3 Power Management 2-5
2.4.4 Registry Settings 2-6
2.4.5 DMA Support 2-8
2.4.6 Unit Test 2-8
2.5 Basic Elements for Driver Development 2-10
2.5.1 BSP Environment Variables 2-11
2.5.2 Mutual Exclusive Drivers 2-11
2.5.3 Dependencies of Drivers 2-11

2.6 Block Device API Reference 2-11
2.6.1 IOCTL_DISK_DEVICE_INFO 2-12
2.6.2 IOCTL_DISK_GET_STORAGEID 2-12
2.6.3 IOCTL_DISK_GETINFO 2-12
2.6.4 IOCTL_DISK_GETNAME 2-13
2.6.5 IOCTL_DISK_READ 2-13
2.6.6 IOCTL_DISK_SETINFO 2-13
2.6.7 IOCTL_DISK_WRITE 2-13
2.6.8 IOCTL_DISK_FLUSH_CACHE 2-14

Chapter 3
Audio Driver

3.1 Audio Driver Summary 3-1
3.1.1 For MX31, MX32 3-1
3.2 Requirements 3-3
3.3 Hardware Operation 3-4
3.3.1 Audio Playback 3-4
3.3.2 Audio Recording 3-6
3.3.3 Required SoC Peripherals 3-7
3.3.4 Conflicts with Other SoC Peripherals 3-7
3.3.5 Known Issues 3-8
3.3.6 Required MC13783 PMIC Components 3-8
3.4 Software Operation 3-8
3.4.1 Audio Playback 3-8
3.4.2 Audio Recording 3-8
3.4.3 Audio Driver Compile-time Configuration Options 3-9
3.4.4 DMA Support 3-11
3.4.5 Power Management 3-12
3.4.6 Audio Driver Registry Settings 3-13
3.5 Unit Test 3-14
3.5.1 Unit Test Hardware 3-15
3.5.2 Unit Test Software 3-15
3.5.3 Building the Audio Driver CETK Tests 3-15
3.5.4 Running the Audio Driver CETK Tests 3-15
3.6 System-level Audio Driver Tests 3-16
3.6.1 Checking for a Boot-time Musical Tune 3-16
3.6.2 Confirming Touchpanel Taps and Keypad Key Presses 3-16
3.6.3 Playing Back Sample Audio and Video Files Using the Media Player 3-17
3.6.4 Using the SDK Sample Audio Applications for Testing 3-17
3.7 Audio Driver API Reference 3-17
3.8 Audio Driver Troubleshooting Guide 3-17
3.8.1 Checking Build-time Configuration Options 3-17
3.8.2 Confirming Audio Driver Loading During Device Boot 3-18
3.8.3 Media Player Application Not Found 3-18

3.8.4 Media Player Fails to Load and Play an Audio File 3-18
3.8.5 No Sound Is Heard During Playback 3-18
3.8.6 Audio Playback Sounds Distorted, Is Too Fast, or Too Slow 3-19

Chapter 4
Backlight Driver

4.1 Smart Backlight Driver 4-1
4.1.1 Backlight Driver Summary 4-1
4.1.2 Requirements 4-2
4.1.3 Hardware Operation 4-2
4.1.4 Software Operation 4-2
4.1.5 Unit Test 4-3
4.1.6 Backlight API Reference 4-3
4.1.7 Power Management 4-7
4.2 LCD Backlight Driver 4-7
4.2.1 Backlight Driver Summary 4-7
4.2.2 Requirements 4-7
4.2.3 Hardware Operation 4-8
4.2.4 Software Operation 4-8
4.2.5 Unit Test 4-9
4.2.6 Backlight API Reference 4-10

Chapter 5
Battery Driver

5.1 Battery Driver Summary 5-1
5.2 Requirements 5-1
5.3 Hardware Operation 5-2
5.3.1 Conflicts with other SoC Peripherals 5-2
5.4 Software Operation 5-2
5.4.1 Battery Driver Registry Settings 5-2
5.5 Unit Test 5-2
5.5.1 Unit Test Hardware 5-3
5.6 Battery API Reference 5-3
5.6.1 Battery PDD Functions 5-3
5.6.2 Battery Driver Structures 5-4
5.7 Power Management 5-4

Chapter 6
Boot from NAND and SD

6.1 Introduction 5-1
6.2 Xloader 5-1
6.2.1 Xloader for NAND Flash 5-1
6.2.2 Xloader for SD/MMC 5-2

6.2.3 Boot Loader for NAND Flash 5-4
6.2.4 Boot Loader for SD/MMC 5-5
6.3 Memory Layout 5-7
6.3.1 NAND Flash Memory Layout 5-7
6.3.2 SD/MMC flash Memory Layout. 5-8
6.3.3 SD/MMC Xloader Layout in Internal RAM. 5-9

Chapter 7
Camera Driver

7.1 Camera Driver Summary 6-1
7.2 Requirements 6-1
7.3 Hardware Operation 6-2
7.3.1 For MX31 and MX32 6-2
7.3.2 Conflicts with other SoC peripherals 6-2
7.4 Software Operation 6-2
7.4.1 Communicating with the Camera 6-2
7.4.2 Camera Registry Settings 6-3
7.4.3 Power Management 6-3
7.5 Unit Test For MX31/MX32 6-4
7.5.1 Unit Test Hardware. 6-4
7.5.2 Unit Test Software 6-4
7.5.3 Building the Camera Tests 6-5
7.5.4 Running the Camera Tests 6-6
7.6 Camera Driver API Reference 6-6
7.6.1 IOCTL_SET_DIRECT_DISPLAY_MODE 6-7

Chapter 8
Chip Support Package Driver Development Kit (ARM11 CSPDDK)

8.1 CSPDDK Driver Summary. 7-1
8.2 Requirements 7-1
8.3 Hardware Operation 7-2
8.3.1 Conflicts with other SoC peripherals 7-2
8.4 Software Operation 7-2
8.4.1 Communicating with the CSPDDK 7-2
8.4.2 Compile-Time Configuration Options 7-2
8.4.3 Registry Settings 7-5
8.4.4 Power Management 7-5
8.5 Unit Test 7-6
8.5.1 Unit Test Hardware. 7-6
8.5.2 Unit Test Software 7-6
8.5.3 Building the Unit Tests 7-6
8.5.4 Running the Unit Tests 7-7
8.6 CSPDDK DLL Reference. 7-7
8.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference 7-7

8.6.2 CSPDDK DLL GPIO (DDK_GPIO) Reference 7-11
8.6.3 CSPDDK DLL IOMUX (DDK_IOMUX) Reference 7-15
8.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference 7-18

Chapter 9
CS8900A Product Ethernet Driver

9.1 CS8900A Product Ethernet Driver Summary 8-1
9.2 Requirements 8-1
9.3 Hardware Operation 8-2
9.3.1 Conflicts with other SoC peripherals 8-2
9.4 Software Operation 8-2
9.4.1 Power Management 8-2
9.4.2 Product Ethernet Registry Settings 8-2
9.5 Unit Test 8-3
9.5.1 Unit Test Hardware 8-3
9.5.2 Unit Test Software 8-4
9.5.3 Building the CS8900A Product Ethernet Tests 8-4
9.5.4 Running the CS8900A Product Ethernet Tests 8-6
9.6 CS8900A Product Ethernet Driver API Reference 8-7

Chapter 10
Configurable Serial Peripheral Interface (CSPI) Driver

10.1 CSPI Driver Summary 9-1
10.1.1 For MX31, MX32 9-1
10.2 Requirements 9-1
10.3 Hardware Operation 9-2
10.3.1 Conflicts with other SoC peripherals 9-2
10.4 Software Operation 9-2
10.4.1 DMA Support (provided with iMX31 and i.MX32) 9-2
10.4.2 Communicating Using the CSPI 9-3
10.4.3 Creating a Handle to the CSPI 9-3
10.4.4 Configuring the CSPI 9-4
10.4.5 Write Operations 9-5
10.4.6 Closing the Handle to the CSPI 9-7
10.4.7 Power Management 9-7
10.4.8 CSPI Registry Settings 9-8
10.5 Unit Test 9-8
10.6 CSPI Driver API Reference 9-9
10.6.1 CSPI Driver IOCTLS 9-9
10.6.2 CSPI Driver Structures 9-9

Chapter 11

Display Driver

11.1 Display Driver Summary 10-1
11.1.1 MX31, MX32ADS 10-1
11.2 Requirements 10-2
11.3 Hardware Operation 10-2
11.3.1 i.MX31 and i.MX32 10-2
11.4 Software Operation 10-3
11.4.1 Communicating with the Display 10-3
11.4.2 Configuring the Display 10-4
11.4.3 Rotation Support 10-5
11.4.4 TV Out Configuration 10-5
11.4.5 Tearing Prevention 10-5
11.4.6 Display Registry Settings 10-5
11.4.7 Power Management 10-7
11.5 Unit Test 10-8
11.5.1 Unit Test Hardware 10-8
11.5.2 Unit Test Software 10-8
11.5.3 Building the Display Tests 10-9
11.5.4 Running the Display Tests 10-10
11.6 Display Driver API Reference 10-13

Chapter 12
Dynamic Voltage and Frequency Control (DVFC)

12.1 DVFC Driver Summary 11-1
12.2 Requirements 11-1
12.3 Hardware Operation 11-2
12.3.1 i.MX31 ADS Configuration 11-2
12.3.2 i.MX32 ADS Configuration 11-2
12.3.3 Conflicts with other SoC peripherals 11-2
12.4 Software Operation 11-2
12.4.1 Loading and Initialization 11-2
12.4.2 i.MX31 Software Operation 11-3
12.4.3 i.MX32 Software Operation 11-5

Chapter 13
Fast Infrared Driver

13.1 FIR Driver Summary 12-1
13.2 Requirements 12-1
13.3 Hardware Operation 12-2
13.3.1 FIR Interface 12-2
13.3.2 Conflicts with other SoC peripherals 12-2
13.4 Software Operation 12-2
13.4.1 DMA Support 12-2

13.4.2 Power Management 12-4
13.4.3 FIR Registry Settings 12-4
13.5 Unit Test 12-4
13.5.1 Unit Test Hardware 12-5
13.5.2 Unit Test Software 12-5
13.5.3 Building the IR Port Tests 12-5
13.5.4 Running the FIR Tests 12-6
13.6 FIR Driver API Reference 12-6

Chapter 14
General Purpose Timer (GPT) Driver

14.1 GPT Driver Summary 13-1
14.1.1 GPT Driver Summary for i.MX31, i.MX32 13-1
14.2 Requirements 13-1
14.3 Hardware Operation 13-2
14.3.1 Conflicts with other SoC peripherals 13-2
14.4 Software Operation 13-2
14.4.1 Communicating with the GPT 13-2
14.4.2 Creating a Handle to the GPT 13-2
14.4.3 Configuring the GPT 13-2
14.4.4 Write Operations 13-3
14.4.5 Closing the Handle to the GPT 13-3
14.4.6 Power Management 13-4
14.4.7 GPT Registry Settings 13-4
14.5 Unit Test 13-4
14.5.1 Unit Test Hardware 13-4
14.5.2 Unit Test Software 13-5
14.5.3 Building the GPT Tests 13-5
14.5.4 Running the GPT Tests 13-5
14.6 GPT Driver API Reference 13-6
14.6.1 GPT Driver Functions 13-6
14.6.2 GPT Driver Structures 13-9

Chapter 15
Hantro Codecs Drivers

15.1 Codecs Drivers Summary 14-1
15.2 Requirements 14-1
15.3 Hardware Operation 14-2
15.3.1 Conflicts with other SoC peripherals 14-2
15.4 Software Operation 14-2
15.4.1 Power Management 14-2
15.4.2 Codecs Registry Settings 14-2
15.5 Unit Test 14-6
15.5.1 Unit Test Software 14-6

15.5.2 Building the Codecs Test Bench . . . 14-6
15.5.3 Customizing the Test Bench/Data . . . 14-7
15.5.4 Running the Codecs Tests . . . 14-8
15.6 Codecs Drivers API Reference . . . 14-8

Chapter 16
Inter-Integrated Circuit (I2C) Driver

16.1 I2C Driver Summary . . . 15-1
16.1.1 For MX31, MX32 . . . 15-1
16.2 Requirements . . . 15-1
16.3 Hardware Operation . . . 15-2
16.3.1 Conflicts with other SoC peripherals . . . 15-2
16.4 Software Operation . . . 15-2
16.4.1 Communicating with the I2C . . . 15-2
16.4.2 Creating a Handle to the I2C . . . 15-2
16.4.3 Configuring the I2C . . . 15-3
16.4.4 Data Transfer Operations . . . 15-4
16.4.5 Closing the Handle to the I2C . . . 15-6
16.4.6 Power Management . . . 15-6
16.4.7 I2C Registry Settings . . . 15-7
16.5 Unit Test . . . 15-7
16.5.1 Unit Test Hardware . . . 15-7
16.5.2 Unit Test Software . . . 15-7
16.5.3 Building the I2C Tests . . . 15-8
16.5.4 Running the I2C Tests . . . 15-8
16.6 I2C Driver API Reference . . . 15-8
16.6.1 I2C Driver IOCTLS . . . 15-8
16.6.2 I2C Driver Macros . . . 15-11
16.6.3 I2C_MACRO_IS_MASTER . . . 15-12
16.6.4 I2C Driver Structures . . . 15-15

Chapter 17
Keypad Driver

17.1 Keypad Driver Summary . . . 16-1
17.1.1 MX31, MX32 Keypad Driver Summary . . . 16-1
17.2 Requirements . . . 16-1
17.3 Hardware Operation . . . 16-2
17.3.1 The MX31 and MX32 ADS Keypad . . . 16-2
17.3.2 Conflicts with other SoC peripherals . . . 16-3
17.4 Software Operation . . . 16-3
17.4.1 MX31 and MX32 Keypad Scan Codes and Virtual Keys . . . 16-3
17.4.2 Power Management . . . 16-5
17.4.3 Keypad Registry Settings . . . 16-6
17.5 Unit Test . . . 16-7

17.5.1 Unit Test Hardware 16-7
17.5.2 Unit Test Software 16-7
17.5.3 Building the Keyboard Tests 16-7
17.5.4 Running the Keyboard Tests 16-7
17.6 Keypad Driver API Reference 16-8
17.6.1 Keypad PDD Functions 16-8

Chapter 18
MBX Direct3D Mobile/OpenGL ES Drivers

18.1 Direct3D Mobile/OpenGL ES Drivers Summary 17-1
18.2 Requirements 17-2
18.3 Hardware Operation 17-2
18.3.1 Conflicts with other SoC peripherals 17-2
18.4 Software Operation 17-2
18.4.1 Communicating with the MBX 17-2
18.5 Configuring the LCD Display Panels 17-3
18.5.1 LCD Display Registry Settings 17-3
18.5.2 Power Management 17-3
18.5.3 Direct3D Mobile and OpenGL ES Registry Settings 17-3
18.6 Unit Test 17-4
18.6.1 Unit Test Hardware 17-5
18.6.2 Unit Test Software 17-5
18.6.3 Building the Direct3D Mobile Tests 17-6
18.6.4 Running the Direct3D Mobile Tests 17-6
18.6.5 Direct3D Mobile/OpenGL ES Application Samples/Demos 17-15
18.7 Drivers API Reference 17-16
18.7.1 Direct3D Mobile 17-16
18.7.2 OpenGL ES 17-16

Chapter 19
NAND Flash Media Driver (FMD)

19.1 NAND FMD Summary 18-1
19.2 Requirements 18-2
19.3 Hardware Operation 18-2
19.3.1 MX31 ADS Configuration 18-3
19.3.2 MX32 ADS Configuration 18-3
19.3.3 Conflicts with other SoC peripherals 18-3
19.4 Software Operation 18-3
19.4.1 Compile-Time Configuration Options 18-3
19.4.2 Loading and Initialization 18-4
19.4.3 Registry Settings 18-4
19.4.4 DMA Support 18-4
19.4.5 Power Management 18-4
19.5 Driver Testing 18-4

19.5.1 Test Hardware 18-4
19.5.2 CETK Testing 18-5
19.5.3 System Testing 18-5

Chapter 20
Notification LED Driver

20.1 Notification LED Driver Summary 19-1
20.2 Requirements 19-1
20.3 Hardware Operation 19-1
20.3.1 Conflicts with other SoC peripherals 19-2
20.4 Software Operation 19-2
20.4.1 Communicating with the Notification LED 19-2
20.4.2 Creating a Handle to the Notification LED 19-2
20.4.3 Configuring the Notification LED 19-3
20.4.4 Closing the Handle of the Notification LED 19-3
20.4.5 Power Management 19-3
20.4.6 Notification LED Registry Settings 19-4
20.5 Unit Test 19-4
20.5.1 Unit Test Hardware 19-4
20.5.2 Unit Test Software 19-4
20.5.3 Building the NLED Tests 19-5
20.5.4 Running the NLED Tests 19-5
20.6 NLED Driver API Reference 19-5
20.6.1 NLED Driver IOCTLS 19-5

Chapter 21
One-Wire Interface (OWIRE)

21.1 One-Wire Interface Driver Summary 20-1
21.2 Requirements 20-1
21.3 Hardware Operation 20-1
21.3.1 Conflicts with other SoC peripherals 20-2
21.4 Software Operation 20-2
21.4.1 Communicating with the One-Wire Interface 20-2
21.4.2 Creating a Handle to the One-Wire Interface 20-2
21.4.3 Configuring the One-Wire Interface 20-2
21.4.4 Write Operations 20-2
21.4.5 Read Operations 20-4
21.4.6 Closing the Handle to the One-Wire Interface 20-6
21.4.7 Power Management 20-6
21.4.8 One-Wire Interface Registry Settings 20-6
21.5 Unit Test 20-7
21.5.1 Unit Test Hardware 20-7
21.5.2 Unit Test Software 20-7
21.5.3 Building the One-Wire Tests 20-7

21.5.4 Running the One-Wire Tests 20-7
21.6 One-Wire Driver API Reference 20-8
21.6.1 One-Wire Driver IOCTLS 20-8
21.6.2 One-Wire Driver Structures 20-9

Chapter 22
PCMCIA Driver

22.1 PCMCIA Driver Summary 21-2
22.2 Requirements 21-2
22.3 Hardware Operation 21-3
22.3.1 Conflicts with other SoC peripherals 21-3
22.4 Software Operation 21-3
22.4.1 Power Management 21-4
22.4.2 PCMCIA Registry Settings 21-5
22.5 Unit Test 21-7
22.5.1 Unit Test Hardware 21-7
22.5.2 Unit Test Software 21-8
22.5.3 Building the PCMCIA Tests 21-8
22.5.4 Running the PCMCIA Tests 21-9
22.6 PCMCIA Driver API Reference 21-10

Chapter 23
Postfilter Driver

23.1 Postfilter Driver Summary 22-1
23.2 Requirements 22-1
23.3 Hardware Operation 22-2
23.3.1 Conflicts with other SoC peripherals 22-2
23.4 Software Operation 22-2
23.4.1 Communicating with the Postfilter Driver 22-2
23.4.2 Creating a Handle to the Postfilter Driver 22-2
23.4.3 Configuring the Postfilter Driver 22-3
23.4.4 Executing Postfilter Operations 22-3
23.4.5 Closing the Handle to the Postfilter Driver 22-4
23.4.6 Postfilter Registry Settings 22-4
23.4.7 Power Management 22-5
23.5 Unit Test For MX31, MX32 22-5
23.5.1 Unit Test Software 22-5
23.5.2 Building the Postfilter Tests 22-6
23.5.3 Running the Postfilter Tests 22-6
23.6 Postfilter Driver API Reference 22-7
23.6.1 Postfilter Driver Functions 22-7
23.6.2 . . . . . PF Driver Enumerations 22-10
23.6.3 . . . . . PF Driver Structures 22-11

Chapter 24
Power Management IC (PMIC)

24.1 PMIC Driver Summary . . . 23-1
24.2 Requirements . . . 23-2
24.2.1 PMIC API Framework . . . 23-2
24.3 Hardware Operation . . . 23-3
24.4 Conflicts with Other SoC Peripherals . . . 23-3
24.4.1 MX31, MX32 Peripheral Conflicts . . . 23-3
24.5 Software Operation . . . 23-3
24.5.1 Configuring the PMIC . . . 23-3
24.5.2 Creating a Handle to the PMIC . . . 23-3
24.5.3 Write Operations . . . 23-4
24.5.4 Read Operations . . . 23-4
24.5.5 Closing the Handle to the PMIC . . . 23-4
24.5.6 Power Management . . . 23-4
24.6 PMIC Registry Settings . . . 23-5
24.7 Unit Test . . . 23-5
24.7.1 Unit Test Hardware . . . 23-5
24.7.2 Unit Test Software . . . 23-5
24.7.3 Building the PMIC Tests . . . 23-5
24.7.4 Running the PMIC Tests . . . 23-6
24.8 PMIC Reference . . . 23-7
24.8.1 PMIC Driver IOCTLS . . . 23-7
24.9 Interrupt Handling . . . 23-9
24.9.1 Interrupt handling Overview . . . 23-9
24.9.2 Interrupt Events . . . 23-10
24.10 Register Access API . . . 23-15
24.10.1 Functions . . . 23-15
24.11 Power Control Reference . . . 23-16
24.11.1 PwCtrl API . . . 23-16
24.11.2 Functions and Data Structures . . . 23-17
24.11.3 PowerCutTimer Functions . . . 23-27
24.11.4 Memory Hold Operation functions . . . 23-28
24.11.5 Power Cut Counter Functions . . . 23-29
24.11.6 Power Management . . . 23-30
24.11.7 Voltage Regulator . . . 23-30
24.11.8 Data Structures . . . 23-30
24.11.9 Switch mode regulator API’s . . . 23-34
24.11.10 Linear Voltage Regulator API’s . . . 23-39
24.11.11 Power Management . . . 23-41
24.11.12 Battery Charger . . . 23-41
24.11.13 Data Structures . . . 23-41
24.11.14 Battery Charger API (Compatible with SC55112 API) . . . 23-41
24.11.15 Battery Charger API (MC13783 Native For Compatibility with SC55112) . . . 23-44

24.11.16 Battery Charger API (MC13783 Native) 23-45
24.11.17 Power Management 23-51
24.12 A/D Converter and Touch. 23-51
24.12.1 Data Types 23-51
24.12.2 Functions. 23-52
24.12.3 Power Management 23-54
24.13 Backlight. 23-54
24.13.1 Data types 23-54
24.13.2 Functions. 23-55
24.13.3 Power Management 23-58
24.14 Connectivity 23-58
24.14.1 Data types 23-58
24.14.2 Functions. 23-61
24.14.3 Power Management 23-68

Chapter 25
Power Manager

25.1 Power Manager Summary 24-1
25.2 Requirements 24-1
25.3 Hardware Operation 24-1
25.4 Software Operation. 24-1
25.4.1 Power Management 24-2
25.4.2 Registry Settings. 24-2
25.5 Unit Test 24-3
25.6 Power Manager API Reference 24-3
25.6.1 Application Interface 24-4
25.6.2 Device Driver Interface. 24-4

Chapter 26
Pulse-Width Modulator (PWM)

26.1 PWM Driver Summary. 25-1
26.2 Requirements 25-1
26.3 Hardware Operation 25-1
26.3.1 Conflicts with other SoC peripherals 25-2
26.4 Software Operation. 25-2
26.4.1 Communicating with the PWM 25-2
26.4.2 Creating a Handle to the PWM. 25-2
26.4.3 Configuring the PWM 25-3
26.4.4 Write Operations. 25-3
26.4.5 Read Operations 25-3
26.4.6 Closing the Handle to the PWM. 25-4
26.4.7 Power Management 25-4
26.4.8 PWM Registry Settings 25-5
26.5 Unit Test 25-5

26.5.1 Unit Test Hardware . . . 25-5
26.5.2 Unit Test Software . . . 25-5
26.5.3 Building the PWM Tests . . . 25-5
26.5.4 Running the PWM Tests . . . 25-6
26.6 PWM Driver API Reference . . . 25-6
26.6.1 PWM Driver IOCTLS . . . 25-6
26.6.2 PWM Driver Macros . . . 25-7
26.6.3 PWM Driver Structures . . . 25-7

Chapter 27
Secure Digital Host Controller

27.1 SDHC Driver Summary . . . 26-1
27.1.1 MX31 & MX32 SDHC Driver Summary . . . 26-1
27.2 Requirements . . . 26-2
27.3 Hardware Operation . . . 26-2
27.3.1 Conflicts with other SoC peripherals . . . 26-2
27.4 Software Operation . . . 26-2
27.4.1 Required Catalog Items . . . 26-2
27.4.2 SDHC Registry Settings . . . 26-3
27.4.3 DMA Support . . . 26-4
27.4.4 Power Management . . . 26-4
27.5 Unit Test . . . 26-5
27.5.1 Unit Test Hardware . . . 26-5
27.5.2 Unit Test Software . . . 26-6
27.5.3 Building the Tests . . . 26-6
27.5.4 Running the Tests . . . 26-6
27.5.5 System Testing . . . 26-8
27.6 Secure Digital Card Driver API Reference . . . 26-9

Chapter 28
Serial Driver

28.1 Serial Driver Summary . . . 27-1
28.1.1 For MX31, MX32 ADS . . . 27-1
28.2 Requirements . . . 27-2
28.3 Hardware Operation . . . 27-2
28.3.1 Conflicts with Other SoC Peripherals . . . 27-2
28.4 Software Operation . . . 27-4
28.4.1 Power Management . . . 27-4
28.4.2 MX31 Serial Registry Settings . . . 27-4
28.4.3 MX32 Serial Registry Settings . . . 27-5
28.4.4 DMA Support . . . 27-7
28.5 Unit Test 1 (Serial Port Driver Tests) . . . 27-8
28.5.1 Unit Test Hardware . . . 27-8
28.5.2 Unit Test Software . . . 27-8

28.5.3 Building the Serial Port Driver Tests 27-8
28.5.4 Running the Serial Port Driver Test 27-8
28.6 Unit Test 2 (Serial Communications Test) 27-9
28.6.1 Unit Test Hardware 27-9
28.6.2 Unit Test Software 27-10
28.6.3 Building the Serial Communications Test 27-10
28.6.4 Running the Serial Communications Test 27-10
28.7 Unit Test 3 (Serial Active Sync Test) 27-13
28.7.1 Unit Test Hardware 27-13
28.7.2 Unit Test Software 27-13
28.7.3 Running Active Sync Test 27-13
28.8 Serial Driver API Reference 27-14
28.8.1 Serial PDD Functions 27-14
28.8.2 Serial Driver Macros 27-15
28.8.3 Serial Driver Structures 27-15

Chapter 29
Slow Infrared Driver

29.1 SIR Driver Summary 28-2
29.1.1 MX31 & MX32 28-2
29.2 Requirements 28-2
29.3 Hardware Operation 28-2
29.3.1 The UART 28-2
29.3.2 Conflicts with other SoC peripherals 28-3
29.4 Software Operation 28-3
29.4.1 Power Management 28-3
29.4.2 MX31 SIR Registry Settings 28-3
29.4.3 MX32 SIR Registry Settings 28-4
29.4.4 DMA Support 28-4
29.5 Unit Test1 (IR Port Tests) 28-5
29.5.1 Unit Test Hardware 28-5
29.5.2 Unit Test Software 28-5
29.5.3 Building the IR Port Tests 28-6
29.5.4 Running the SIR Tests 28-6
29.6 SIR Driver API Reference 28-10
29.6.1 SIR PDD Functions 28-10
29.6.2 SIR Driver Macros 28-11
29.6.3 SIR Driver Structures 28-11

Chapter 30
Subscriber Identification Module (SIM) Driver

30.1 SIM Driver Summary 29-1
30.2 Requirements 29-1
30.3 Hardware Operation 29-2

30.3.1 Conflicts with Other SoC Peripherals . . . 29-2
30.4 Software Operation . . . 29-2
30.4.1 Power Management . . . 29-2
30.4.2 SIM Registry Settings . . . 29-2
30.5 Unit Test . . . 29-3
30.5.1 Unit Test Hardware . . . 29-3
30.5.2 Building the SIM Application . . . 29-3
30.5.3 Running the SIM Application . . . 29-3
30.6 SIM Driver API Reference . . . 29-4
30.6.1 SIM PDD Functions . . . 29-4
30.6.2 SIM Driver Macros . . . 29-4
30.6.3 SIM Driver Structures . . . 29-4

Chapter 31 Touch Panel Driver

31.1 Touch Panel Driver summary . . . 30-1
31.2 Requirements . . . 30-1
31.3 Hardware Operations . . . 30-1
31.4 Conflicts with SOC peripherals . . . 30-2
31.4.1 Conflicts with iMX31, iMX32 . . . 30-2
31.5 Software Operations . . . 30-2
31.5.1 Touch driver registry settings . . . 30-2
31.6 Unit Tests . . . 30-3
31.6.1 Unit test Hardware . . . 30-3
31.6.2 Unit test software . . . 30-3
31.6.3 Building the touch panel Tests . . . 30-3
31.7 Touch Panel API reference . . . 30-4

Chapter 32 USB OTG Driver

32.1 USB OTG Driver Summary . . . 31-1
32.1.1 OTG Client Driver Summary . . . 31-1
32.1.2 OTG Host Driver Summary . . . 31-2
32.1.3 OTG Transceiver Driver Summary (For HIGH-SPEED only) . . . 31-3
32.2 Requirements . . . 31-3
32.3 Hardware Operation . . . 31-4
32.3.1 Conflicts with other Peripherals . . . 31-4
32.4 Software Operation . . . 31-5
32.4.1 USB OTG Host Controller Driver . . . 31-5
32.4.2 USB Client Driver . . . 31-15
32.4.3 USB Transceiver Driver (ID Pin Detect Driver -- XCVR) . . . 31-19
32.4.4 Power Management . . . 31-24
32.4.5 Function Drivers . . . 31-27
32.4.6 Class Drivers . . . 31-29
32.5 USB High Speed Host (H2) Driver . . . 31-30

32.5.1 USB HS Host 2 (H2) Driver Summary 31-30
32.5.2 Class Drivers 31-30
32.5.3 Requirements 31-31
32.5.4 Hardware Operation 31-32
32.5.5 Software Operation 31-32
32.6 USB Full Speed Host (H1) Driver 31-34
32.6.1 USB Full Speed Host (H1) Driver Summary 31-34
32.6.2 Requirements 31-34
32.6.3 Hardware Operation 31-35
32.6.4 Software Operation 31-35
32.7 Basic Elements for Driver Development 31-37
32.7.1 BSP Environment Variables 31-37
32.7.2 Dependencies of Drivers 31-38

Chapter 33
Video Processing Unit (VPU)

33.1 VPU driver summary 32-1
33.2 Requirements 32-1
33.3 Hardware Operation 32-2
33.3.1 Conflicts with other SoC peripherals 32-2
33.4 Software Operation 32-2
33.4.1 Communicating with the VPU 32-2
33.4.2 Power Management 32-2
33.4.3 Codecs Registry Settings 32-2
33.5 Unit Test 32-2
33.5.1 Unit Test Hardware 32-3
33.5.2 Unit Test Software 32-3
33.5.3 Running the VPU Application Test 32-3
33.6 VPU Driver API Reference 32-3
33.7 Sample Application 32-4
33.7.1 System Requirements on MX32 32-4
33.7.2 Building the WinCE Image and VPU Test Application 32-4

About This Book

This document is a reference guide for the Freescale Windows CE 5.0 board support package (BSP). The contents of this document will describe the requirements, implementation and testing for each of the drivers included in this BSP. Information on how to use the custom stream interface drivers is also provided.

Audience

This guide is intended for users of the Freescale Windows CE 5.0 BSP that need detailed driver information. The information included in this guide can be used in preparation to support another hardware platform, write applications and develop and run tests. The audience includes device driver developers, application developers and test engineers.

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

ADS	application development system
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
I2C	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
NLED	Notification Light Emitting Diode
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
SIM	subscriber identification module
SIR	slow infrared
SOC	system on a chip
UART	universal asynchronous receiver transmitter
USB	universal serial bus

Suggested Reading

Additional information regarding Microsoft Windows CE and the Freescale platform can be found in these documents:

- http://msdn.microsoft.com/embedded/windowsce
- HW Getting Started – MX31 ADS

- MC9328MX31 Applications Processor IC Reference Manual
- Windows CE BSP for MX31 User's Guide
- MC13783 Detailed Technical Specification (L3)

References

The following documents were referenced to produce this document.

- Platform Builder for Microsoft Windows CE 5.0 Help
- HW Getting Started – MX31 ADS
- Windows CE BSP for MX31 User's Guide
- MC13783 Detailed Technical Specification (L3)

Chapter 1
Introduction

1.1 Overview

This Board Support Package (BSP) supports the following Freescale hardware reference platforms:

- i.MX31ADS—Windows CE 5.0
- i.MX32ADS—Windows CE 5.0

1.2 BSP Scope

The purpose of this BSP software package is to support the Windows CE 5.0 operating system on the MX31ADS and MX32ADS platforms. It provides the software necessary to interface the Windows CE 5.0 operating system to the hardware. The goal of this BSP is to enable Freescale customers to rapidly build products based on the hardware using the Windows CE operating system.

The BSP is not a platform or product reference implementation. It does not contain all of the product specific drivers, hardware independent software stacks, applications, etc. required for a product. End customers must customize and optimize this BSP for their own hardware platform.

The BSP implements the Production Quality OEM Abstraction Layer (PQOAL) and Production Quality Drivers (PQD) software architecture. The PQOAL and PQD architecture reduces the amount of code the OEM needs to port for designs based on Freescale processors.

The BSP is not intended to be used for silicon verification. While it can play a role in this, the BSP functionality and the tests run on the BSP do not have sufficient coverage to replace silicon verification test suites.

1.3 Features

The features included in this release are documented in the BSP *release notes*. A separate release notes document is provided for each platform. Please study the release notes for the appropriate platform for details.

1.4 Getting Started

For instructions on installing the BSP, building, downloading and running the OS image on the hardware board, refer to the BSP User's Guide included with this distribution.

1.5 Windows CE Architecture

The Windows CE architecture is documented in the Platform Builder help. The architecture of the operating system and sub-systems (e.g. power management, DirectDraw, etc.) are in various locations in help. See the following location as a starting point on the Windows CE architecture: **Welcome to Windows CE 5.0 –> Windows CE Architecture.**

Chapter 2
ATA Driver

ATA driver in WinCE 5.0 is a block driver, used as the lower layer for File Systems and USB mass storage, for example. It is constructed as a stream interface driver that exposes I/O control codes (IOCTL_DISK_XXX and DISK_IOCTL_XXX). The file system uses these I/O control codes to access the ATA devices.

2.1 ATA Driver Summary

2.1.1 MX31 and MX32

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\ATA
CSP Static Library	ata_<*TGTSOC*>.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\BLOCK\ATA
Import Library	(cspddk.lib)
Driver DLL	ata_<*TGTSOC*>.dll
Catalog Item	Third Party —> BSPs —> Freescale <*TGTPLAT*>: ARMV4I —> Storage Drivers —> ATA device driver
SYSGEN Dependency	SYSGEN_MSPART,SYSGEN_FATFS
BSP Environment Variable	BSP_ATA

2.2 Requirements

2.2.1 MX31 and MX32

The ATA driver must meet/support the following requirements:

1. Provide standard Microsoft Block Storage Device API, including disk information management, formatting, block data read/write with full scatter-gather buffer support
2. Support two power management modes, full on and full off

3. Support standard bus timing modes for PIO modes 0-4, MDMA modes 0-2, and UDMA modes 0-5 (performance tested practically to UDMA mode 4 only)
4. Support full sustained (media) data throughput capacity of Hitachi TravelStar C4K40 (or equivalent) at UDMA mode 3 in MX31 ADS.
5. Support full sustained (media) data throughput capacity of Hitachi TravelStar C4K40 (or equivalent) at UDMA mode 4 in MX32 ADS.

2.3 Hardware Operation

Refer to the chapter on the Advanced Technology Attachment (ATA) in the hardware specification document for detailed operation and programming information.

The MX31 or MX32 contains an on-chip ATA controller. Data transfers on the ATA bus can take place through:

NOTE

CPU programmed data transfers via ATA controller registers. (Programmed I/O modes, “PIO” modes 0-4)

“Multi-word” DMA (MDMA modes 0-2)

“Ultra” DMA (UDMA modes 0-5)

Within the types of ATA-bus data transfer (PIO or xDMA), the various “modes” (0-*n*) refer only to specified combinations of timing parameters, as supported by industry standard hardware. The ATA DMA modes transport data between the ATA peripheral (disk) and the system bus, via the MX31 or MX32's ATA peripheral data FIFO.

2.3.1 ATA Hardware Block Diagram

2.3.1.1 MX31 and MX32

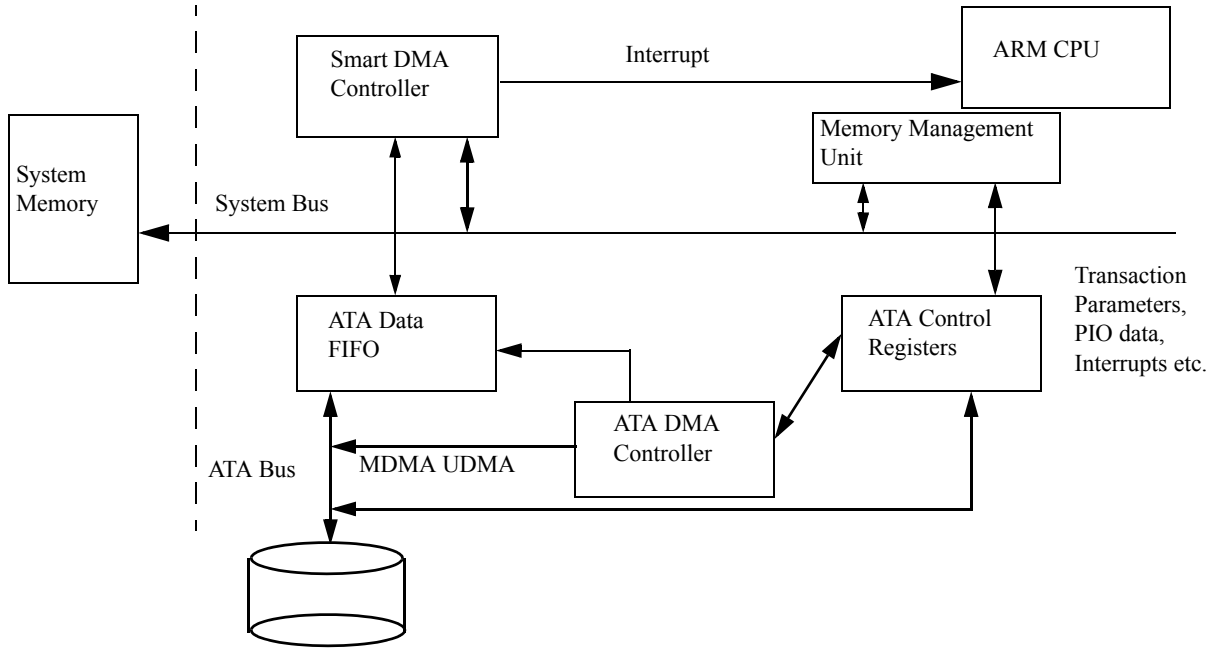


Figure 2-1. ATA Hardware Block Diagram

The MX31 and MX32 contain a “Smart DMA controller” (SDMA) which acts as a third-party bus mastering DMA, for transporting data between the ATA data FIFO and system memory. SDMA support is built in to the ATA driver, and is automatically configured and used when UDMA or MDMA modes are selected for data transport on the ATA bus. The default block/sector size is 512 bytes. With these sector sizes, far greater efficiency in processor/bus usage is gained by setting UDMA or MDMA modes, instead of PIO modes. The PIO modes are provided for functional compatibility with legacy hardware which may not support fastest current data rates.

The appropriate ATA-specific mode (PIO, MDMA or UDMA) must be selected based on the capabilities of the specific attached ATA peripheral.

2.3.2 Conflicts with other Peripherals and Catalog Options

2.3.2.1 Conflicts with SoC Peripherals

2.3.2.1.1 MX31 and MX32

On the MX31 and MX32 processor, the ATA has signals which can conflict with the CSPI device (CSPI1), USB Host (1& 2) ports and PWM module depending on configuration. See below for details of current implementation.

The MX31 and MX32 processor can be configured in modes where ATA and USB Host ports do not share pins. Only ATA and USB Host 2(High Speed) port configuration is supported in current design, but it is not stable, UDMA3 or PIO2 mode will failed in CETK Storage Device Block Driver API Test cases, so if ATA UDMA3 or PIO2 mode are used, the USB Host 2(High Speed) port should not be configured in the image. Due to hardware limitation ATA and USB Host 1(Full Speed) port configuration is not supported in MX31 and MX32 ADS.

2.3.2.2 Conflicts in ADS evaluation board

2.3.2.2.1 MX31 and MX32

The ATA as implemented for the MX31, MX32/ADS evaluation board has pin conflicts with the CSPI device (CSPI1), USB Host (1 & 2) ports and PWM. When the ATA is configured in the BSP (selected from the catalog), none of CSPI1, USB Host 1 and PWM can be configured in the image.

The positions of these devices in the catalog:

Third Party \ Freescale MX31: ARMV4I \ Device Drivers \ CSPI Bus

Third Party \ Freescale MX31: ARMV4I \ Device Drivers \ USB Devices \ USB Host Devices\USB Full Speed Host 1

Third Party \ Freescale MX31: ARMV4I \ Device Drivers \ PWM

2.3.3 Cabling

The ATA specification requires an 80 core ribbon cable when used in UDMA modes 3 or greater. This may be relaxed for cables shorter than the maximum defined in the specification.

2.4 Software Operation

2.4.1 Application / User Interface to ATA drives

The ATA device exports a standard streams interface to the Windows File System. Application-level access to ATA disks is via the Windows File System, using functions such as CreateFile() and CloseHandle().

The File System, or user software which requires block device access to the ATA, does so through the standard Windows CE Block Device IOCTLs as described in section 0. These provide functions to acquire disk information and to read and write blocks (disk sectors) of data.

2.4.2 ATA Driver Configuration

The driver is configured into the BSP build by dragging & dropping the catalog item as defined in 2.1.

This defines the environment variable/configuration option: BSP_ATA.

Configuration for the ATA is then provided through registry settings imported from platform.reg. These settings can be modified to select timing and transfer mode, and if necessary the device prefix and index.

2.4.2.1 Transfer Mode and Timing

The mode by which data is transported on the ATA bus (TransferMode) is configured by a registry setting defined in section 2.4.2.4.

2.4.2.1.1 MX31 and MX32

The ATA bus timings are based on the i.MX31 and i.MX32 clock, as defined in the MX31 hardware reference manual. The driver requires a clock period of 15 nanosec (66.6MHz).

2.4.2.2 Prefix and Index

The default device prefix is “DSK” and Index is “1”. These items are important when configuring a storage device as source for the USB Mass Storage client. The USB Mass Storage client (function) driver's default registry configuration, from PUBLIC\Common\OAK\FILES\common.reg, sets the source block device as “DSK1”.

When no Index is configured for the ATA block device, the bus enumerator will assign an index according to the order of block device loading. When removable storage is attached to USB host ports (as mass storage class), or when RAMDISK is included, the index assigned to these other block devices can influence any Index automatically assigned by the bus enumerator.

2.4.2.3 IOMUX and Pinout

2.4.2.3.1 MX31 and MX32

The internal MX31 and MX32 ATA signals can be multiplexed to a choice of pins on IC, as described for the IOMUX in the hardware reference manual. The ADS also supports multiplexing these ATA signals through a choice of buffers in one of two configurations.

The ATA driver is currently supports two hardware configuration, which defines the other on-chip peripherals and devices with which an ATA device can be simultaneously configured within the BSP.

2.4.2.4 Defaults

The default mode for the ATA is transfer mode UDMA mode 3 for MX31 and UDMA mode 4 for MX32, as selected by the default platform.reg file supplied for the build.

Default IOMUX configuration supports hardware mode 1. (This has implications for peripheral conflicts as per section 2.3.1).

2.4.3 Power Management

The ATA supports two power management modes, ON (D0) and OFF (D4). These modes are managed via the standard Windows Power Manager. Power Manager uses IOCTL_POWER_SET to switch the disk's power state, according to inactivity settings configured in Power Manager.

As for standard block drivers, PowerUp and PowerDown functions are called by the Device Manager.

The primary method for limiting power consumption in the ATA module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the DDKClockSetGatingMode function call. The clock is turned on during initialization process and the clock is turned off after initialization is completed. Data transfer operations are handled in DSK_IOCTL function to process the IOCTL calls from the File System. The ATA driver turns ON the clock and enables the ATA module before processing any data transfer. After the block of data has been processed, the ATA module is disabled and the clock is turned OFF.

2.4.3.1 PowerUp

This function called by Device Manager sets a flag to indicate power is up.

2.4.3.2 PowerDown

This function called by Device Manager ensures volatile data is stored in RAM and sets a flag to indicate power is down.

2.4.3.3 IOCTL_POWER_SET

This IOCTL handles the request to change disk power state (D0 or D4), called by Power Manager (or SetDevicePower() API).

2.4.4 Registry Settings

The ATA driver settings are taken from platform.reg, which can be customized for each particular build. These registry values are located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ATA_MX31] for MX31
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ATA_MX31] for MX32
```

The values under that registry key should be defined in platform.reg. These can be qualified with the BSP_ATA system variable for configurable catalog item support.

Value	Type	Content	Description
Dll	sz	ata_mx31.dll for MX31, ata_mx32.dll for MX32	Driver dynamic link library
IClass	sz	"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}" GUID for a power-manageable block device	
TransferMode	dword	08	PIO mode 0
		09	PIO mode 1
		...	...
		0C	PIO mode 4
		20	MDMA mode 0
		21	MDMA mode 1

Value	Type	Content	Description
		22	MDMA mode 2
		41	UDMA mode 1
		...	...
		45	UDMA mode 5
IOMuxMask	dword	040007f8 (Only used in MX31)	GPR mask for IOMUX setting. Derived from MX31 hardware manual and devices configured in combination
IOMuxVal	dword	00000318 (Only used in MX31)	GPR value for IOMUX setting, as above.
InterruptDriven	dword	01 (00)	enable interrupt driven I/O use for PIO or MDMA/UDMA modes (disable interrupt; not used normally)
DMA	dword	00 01	disable DMA (always disable for PIO mode) enable DMA (always enable for MDMA or UDMA modes)
IORDYEnable	dword	01	enable Host IORDY for PIO mode 3 and 4

As indicated in the above table, the following settings should be combined:

For PIO modes:

```
"InterruptDriven"=dword:01      ; 01-enable interrupt driven I/O, 00-disable
"DMA"=dword:00                  ; disable DMA
"TransferMode"=dword:0c         ; 08-PIO mode 0, ..., 0C-PIO mode 4
"IORDYEnable"=dword:01          ; enable Host IORDY for PIO mode 3, 4
```

For MWDMA modes:

```
"InterruptDriven"=dword:01      ; enable interrupt driven I/O
"DMA"=dword:01                  ; enable DMA
"TransferMode"=dword:20         ; 20-MWDMA mode 0, ..., 22-MWDMA mode 2
"IORDYEnable"=dword:01          ; enable Host IORDY for PIO mode 3, 4
```

For UDMA modes:

```
"InterruptDriven"=dword:01      ; enable interrupt driven I/O
"DMA"=dword:01                  ; enable DMA
"TransferMode"=dword:42         ; 40-UDMA mode 0, ..., 45-UDMA mode 5
"IORDYEnable"=dword:01          ; enable Host IORDY for PIO mode 3, 4
```

Standard registry entries also to be included for the ATA device under the above key, are as below:

Value	Type	Content	Description
Prefix	sz	“DSK”	Device identifier (combined with Index for DSK1 for example)
Index	dword	1	Instance of ATA drive.
Order	dword	10	Early, to allow file system loading
DoubleBufferSize	dword	10000	128 sectors
DrqDataBlockSize	dword	200	Each data request is one sector, always 512 bytes
WriteCache	dword	01	disk internal cache is enabled within drive

Value	Type	Content	Description
LookAhead	dword	01	disk read-ahead to internal is enabled within drive
DeviceId	dword	00	primary device ID
HDProfile	sz	“HDProfile”	Storage Manager profile to be used in GetDeviceInfo (see below)

In addition to these values, the ATA makes use of the HDProfile information from the StorageManager registry keys. Default/sample values are as below:

```
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\HDProfile]
"Name"="ATA Hard Disk Drive"
"Folder"="Hard Disk"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\HDProfile\FATFS]
"EnableCacheWarm"=dword:00000000
```

2.4.5 DMA Support

ATA driver supports DMA mode and non-DMA mode of transfer.The driver defaults to DMA mode of transfer always.ATA supports three transfer-types : UDMA, MDMA and PIO mode. PIO mode works in Non-DMA mode of operation while other modes works in DMA mode. To change the mode of tranfer, we have to change the value of “TransferMode“ from the registry. When ATA driver operates in SDMA, it always uses the scatter gather method. Though the flag BSP_SDMA_SUPPORT_ATA is present in bsp_cfg.h, it does not control whether SDMA is used or not.

The driver does not allocate or manage DMA buffers internally. All buffers are allocated and managed by the upper layers, the details of which are given in the request submitted to the driver. For every request submitted to it, the driver attempts to build a DMA Scatter Gather Buffer Descriptor list for the buffer passed to it by the upper layer.

For the driver to attempt to build the Scatter Gather DMA Buffer Descriptors, the upper layer should ensure that the buffer meets the following criteria.

- Start of the buffer should be a cache-line (32byte) aligned address.
- Number of bytes to transfer should be cache-line (32byte) aligned.

2.4.6 Unit Test

The ATA driver is tested using the Storage Block Device test cases included as part of the Windows CE 5.0 Test Kit (CETK). There are no custom CETK test cases for ATA driver. The Storage Device test cases used to test ATA driver include:

1. File System Driver Test cases
2. Storage Device Block Driver API Test cases
3. Storage Device Block Driver Read/Write Test cases
4. Storage Device Block Driver Benchmark Test cases

2.4.6.1 Unit Test Hardware

The following table lists the required hardware to run the ATA driver unit tests.

2.4.6.1.1 MX31

Requirements	Description
i.MX31 and attached HITACHI hard disk C4K40.	Other drives supporting up to UDMA mode 3 may be used. Cable connection must be <15 cm or use 80-core UDMA3+ cable.

2.4.6.1.2 MX32

Requirements	Description
i.MX32 and attached HITACHI hard disk C4K40.	Other drives supporting up to UDMA mode 4 may be used. Cable connection must be <15 cm or use 80-core UDMA3+ cable.

2.4.6.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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	Test .dll file used to perform File System Driver Test cases.
disktest.dll	Test .dll file used to perform Storage Device Block Driver API test cases.
rw_all.dll	Test .dll file used to perform Storage Device Block Driver Benchmark tests.
rwtest.dll	Test .dll for various read/write options, including multi-threading and various block sizes.

2.4.6.3 Building the Storage Device Tests

The Storage Device Tests come pre-built as part of the CETK. No steps are required to build these tests. The fsdtst.dll, disktest.dll rw_all.dll and rwtest.dll files can be found alongside the other required CETK files in the following location:

```
\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I
```

2.4.6.4 Running the Storage Device Tests

These CETK tests will destroy any information residing on the hard disk.

The tests can be launched from command line or CE Target Control window in Platform Builder. The command line for running the File System Driver Test is:

```
tux -o -d fsdtst  -x 1001-1010,5001-5031 -c "-p HDProfile -zorch”
```

This performs file system tests which cover all required File System API functions. Excluded are those tests which manipulate disk partitions.

The command line option HDProfile refers to the registry setting used to establish storage device profile information to the Storage Manager:

```
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\HDProfile]
"Name"="ATA Hard Disk Drive"
"Folder"="Hard Disk"
```

NOTE

The command line option "-zorch" is case sensitive (help message within the test .dll is not correct) and is used to confirm over-writing of all information on the hard disk.

The command line for running the Storage Device Block Driver API Test is:

```
tux –o –d disktest –c  "-p HDProfile -zorch"
```

The command line for running the Storage Device Block Driver Read/Write Test is:

```
tux -o -d rwtest -c "-p HDProfile -zorch"
```

The command line for running the Storage Device Block Benchmark Test is:

```
tux -o -d rw_all –x 1006 –c "-p HDProfile -zorch"
```

This includes only the benchmark test for 128 contiguous sectors. The test reads and writes all sectors of the drive in 128 block (64 kByte) chunks. When drive read-ahead is enabled, this will allow drive to provide maximum sustained data rate from the media, to ensure ATA driver supports the same. It is not necessary for all drive sectors to be tested, but the pre-compiled test does not have options to limit the portion tested, and all components are not publicly available for test customization. The test takes approximately 4 hours to execute on a 40 GB drive. Tests using smaller contiguous chunks take even longer, and are not necessary for driver characterization.

For detailed information on the Storage Device CETK test cases, please refer to:

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> File System Driver Test

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Storage Device Block Driver API Test

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Storage Device Block Driver Read/Write Test

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Storage Device Block Benchmark Test

2.5 Basic Elements for Driver Development

This chapter provides details of the basic elements for driver development in the <TGTPLAT> BSP.

2.5.1 BSP Environment Variables

Names	Definition
BSP_ATA	Set to enable ATA device driver

NOTE

For MX31 and MX32, it is possible to configure ATA with USBH2.

2.5.2 Mutual Exclusive Drivers

2.5.2.1 MX31 and MX32

Some conditions to set environment variables for mutual exclusive drivers (a driver can not co-exist with another driver) are listed below.

Environment Variables	Reason
BSP_USB_FSH1 and BSP_ATA can not be set '1' at same time	Pin conflicts between Host 1 and ATA mean they can not coexist. It is hardware limitation for the MX31 board.
BSP_CSPI1 and BSP_ATA cannot be set '1' at same time	Pin conflicts between CSPI1 device and mean they can not coexist.
BSP_PWM and BSP_ATA can not be set ‘1’ at same time	Pin conflicts between PWM module and ATA mean they can not coexist

2.5.3 Dependencies of Drivers

The following table summarizes the Microsoft defined environment variables used in the BSP.

Names	Definition
SYSGEN_FATFS	Set to support FAT16 file system
SYSGEN_STOREMGR	Set to support storage manager
SYSGEN_STOREMGR_CPL	Set to support storage manager in control panel
SYSGEN_MSPART	Set to support partition driver.

2.6 Block Device API Reference

The primary interface to the ATA block device is via the standard Windows CE Block Device IOCTLs as described in the following sections. Application-level access to ATA disks should be through the Windows File System.

For reverse compatibility deprecated DISK_IOCTL* are also supported but not documented here. See CE 5.0 Help for further details.

The driver also supports the standard XXX_Init, XXX_Deinit, XXX_Open and XXX_Close routines, as used by Device Manager and the bus enumerator to load the driver. When the registry settings for ATA are correct, these functions are handled automatically, and need no further documentation here.

2.6.1 IOCTL_DISK_DEVICE_INFO

This **DeviceIoControl** request returns storage information to block device drivers.

Parameters

lpInBuffer	[in] Pointer to a STORAGEDEVICEINFO structure.
nInBufferSize	[in] Specifies the size of the STORAGEDEVICEINFO structure.
lpBytesReturned	[out] Pointer to a DWORD to receive the total number of bytes returned.

2.6.2 IOCTL_DISK_GET_STORAGEID

This DeviceIoControl request returns the current STORAGE_IDENTIFICATION structure for a particular storage device.

Parameters

hDevice	[in] Handle to the block device storage volume, which can be obtained by opening the FAT volume by its file system entry. The following code example shows how to open a PC Card storage volume. hVolume = CreateFile(TEXT("\Storage Card\Vol:"), GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
lpOutBuffer	[out] Set to the address of an allocated STORAGE_IDENTIFICATION structure. This buffer receives the storage identifier data when the IoControl call returns
nOutBufferSize	[out] Set to the size of the STORAGE_IDENTIFICATION structure and also additional memory for the identifiers. For Advanced Technology Attachment (ATA) disk devices, the identifiers consist of 20 bytes for a manufacturer identifier string, and also 10 bytes for the serial number of the disk.
lpBytesReturned	[out] Pointer to a DWORD to receive the total number of bytes returned.

2.6.3 IOCTL_DISK_GETINFO

This DeviceIoControl request returns notifies the block device drivers to return disk information.

Parameters

lpOutBuffer	[out] Pointer to a DISK_INFO structure.
nOutBufferSize	[out] Specifies the size of the DISK_INFO structure.
lpBytesReturned	[out] Pointer to a DWORD to receive the total number of bytes returned.

2.6.4 IOCTL_DISK_GETNAME

This DeviceIoControl request services the request from the FAT file system for the name of the folder that determines how users access the block device. If the driver does not supply a name, the FAT file system uses the default name passed to it by the file system.

Parameters

lpOutBuffer [out] Specifies a buffer allocated by the file system driver. The device driver fills this buffer with the folder name. The folder name must be a Unicode string.

nOutBufferSize [out] Specifies the size of lpOutBuffer. Always set to MAX_PATH where MAX_PATH includes the terminating NULL character.

lpBytesReturned [out] Set by the device driver to the length of the returned string and also the terminating NULL character.

2.6.5 IOCTL_DISK_READ

This DeviceIoControl request services FAT file system requests to read data from the block device.

Parameters

lpInBuffer [in] Pointer to a SG_REQ structure.

nInBufferSize [in] Specifies the size of the SG_REQ structure.

lpBytesReturned [out] Pointer to a DWORD to receive total bytes returned. Set to NULL if you do not need to return this value.

2.6.6 IOCTL_DISK_SETINFO

This DeviceIoControl request services FAT file system requests to set disk information.

Parameters

lpInBuffer [in] Pointer to a DISK_INFO structure.

nInBufferSize [in] Specifies the size of DISK_INFO.

lpBytesReturned [out] Pointer to a DWORD to receive total bytes returned.

2.6.7 IOCTL_DISK_WRITE

This DeviceIoControl request services FAT file system requests to write data to the block device.

Parameters

lpInBuffer [in] Pointer to an SG_REQ structure.

nInBufferSize [in] Specifies the size of SG_REQ.

lpBytesReturned [out] Pointer to a DWORD to receive total bytes returned.

See the sr_status member of SG_REQ for write status. ERROR_SUCCESS indicates write success.

2.6.8 IOCTL_DISK_FLUSH_CACHE

This DeviceIoControl request issues the ATA FLUSH CACHE command to the disk.

Parameters [No parameters]

Return Value ERROR_SUCCESS: flushed okay

ERROR_GEN_FAILURE: Failed to send flush command. Either write caching was not enabled on the device, or command was aborted.

Chapter 3
Audio Driver

The audio driver module for the MC13783 PMIC (wavedev_MC13783.dll) is used for providing audio playback and recording functions. This module is capable of performing audio playback using the MC13783 PMIC Stereo DAC and recording using the MC13783 Voice CODEC. Full duplex operation, that is, simultaneous playback and recording is also supported.

Playback support is always enabled. However, recording support can be explicitly enabled or disabled by the use of a BSP variable (see Section 3.1.1, "For MX31, MX32"). When Recording is disabled the driver will work in half duplex mode. This feature is currently supported for both the MX31 and MX32.

An application can access the audio driver by using any of the methods and functions that are described in the following Platform Builder online help section:

Windows CE Features → Graphics and Multimedia Technologies → Audio → Waveform Audio → Waveform Audio Application Development

3.1 Audio Driver Summary

The table below provides a summary of the source code location, library dependencies, and other BSP information:

3.1.1 For MX31, MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\AUDIODEV ..\CSP\ARM\FREESCALE\PMIC\MC13783\SDK
IC-specific CSP Driver Path	N/A
CSP Static Library	mxarm11_wavedev_record.lib (When recording is enabled.) mxarm11_wavedev_record_stubs.lib (When recording is disabled.) mxarm11_wavedev.lib pmicSdkCsp_MC13783.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*> \SRC\DRIVERS\PMIC\MC13783\PDK ..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\WAVEDEV\MC13783
Import Library	N/A
Driver DLL	wavedev_MC13783.dll

Driver Attribute	Definition
Required Catalog Items	Third Party → BSPs → Freescale <*TGTPLAT*> → Device Drivers → Audio → MC13783 Audio Driver
Recommended Catalog Items	Core OS → Windows CE devices → Graphics and Multimedia Technologies → Audio → Waveform Audio Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Audio Codecs and Renderers → MP3 Codec Core OS → Windows CE devices → Graphics and Multimedia Technologies →Media → Audio Codecs and Renderers → MPEG-1 Layer 1 and 2 Audio Codec Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Audio Codecs and Renderers → MS ADPCM Audio Codec Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Audio Codecs and Renderers → Wave/AIFF/au/snd File Parser Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Audio Codecs and Renderers → Waveform Audio Renderer Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Audio Codecs and Renderers → WMA Codec Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Windows Media Player → Windows Media Player Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Windows Media Player → Windows Media Player OCX Core OS → Windows CE devices → Graphics and Multimedia Technologies → Media → Windows Media Player → Windows Media Technologies
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_AUDIO_MC13783=1 BSP_PMIC_MC13783=1 BSP_AUDIO_RECORDING=1 To include recording support BSP_AUDIO_RECORDING=0 To remove recording support Note: If BSP_AUDIO_RECORDING is not defined at all, then recording support will not be included.

The Recommended Catalog Items listed in Table 1 should be included in the OS design in order to provide a fairly comprehensive audio playback capability using the Windows Media Player application. The choice of which audio CODECs to include or exclude from the OS design can be altered based upon the specific functional requirements and degree of audio support that is desired.

Note that the selection and use of the Windows Media Player and the various software CODECs is beyond the scope of the audio driver and will not be discussed further in this document. Please refer to the following Platform Builder online help section if additional information about these items is required:

Windows CE Features → Graphics and Multimedia Technologies

3.2 Requirements

The audio driver must satisfy all of the following requirements:

1. The driver shall conform to the Microsoft audio driver architecture as defined for WinCE and all related operating systems.
2. The driver shall support any Freescale MXARM11 based platform that is compatible with the MC13783 PMIC.
3. The driver support audio recording using the Microsoft-documented WAV APIs.
4. The driver shall use double-buffered DMA operations to transfer audio data between memory and the SSI FIFO.
5. The driver shall support two power management modes, full on and full off.
6. The driver shall minimize power consumption at all times by using clock gating and by disabling all audio-related hardware components that are not actively being used.

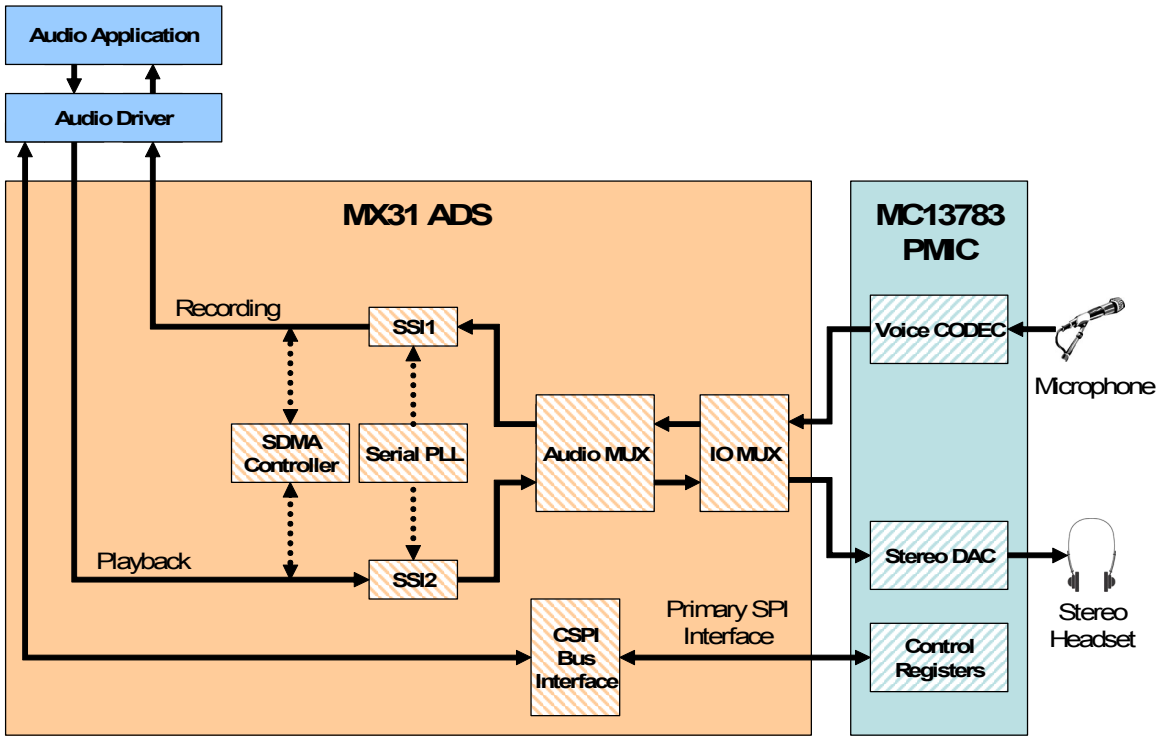


Figure 3-1. Audio Playback and Recording Hardware Components

3.3 Hardware Operation

Figure 3-1 shows the hardware components and the default configuration that is used for both audio playback and recording. Refer to the chapters in the IC-specific Reference Manual for the SSI, Serial Clock PLL, SDMA, Audio MUX, and IO MUX components for detailed operation and programming information. Also refer to the MC13783 DTS document for complete technical details concerning all of the MC13783 audio components. This includes the Stereo DAC, the Voice CODEC, the various audio input/output paths that are available, and the supported amplifier/mixer configurations.

The schematics for the platform and the MC13783 PMIC (which may be in the form of an add-on daughter card) should also be consulted if information about the routing of the various audio-related signal lines is needed.

Also see the Audio Driver Compile-time Configuration Options section below for information about how to change or fine-tune the hardware configuration for audio playback and recording.

3.3.1 Audio Playback

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

- Configure SSI2 for time-slotted network mode using 4 timeslots/frame and a sampling rate of 44.1 kHz. The first two timeslots are used to transmit the left and right audio channel data words, respectively. Each audio data word is 16 bits long. SSI2 is also configured to operate in master mode using the Serial PLL (for i.MX31) clock signal to generate the appropriate framesync and bitclock signals. For the i.MX32 the default build configuration has SSI2 operating in slave mode while the MC13783 PMIC is the master. Note that we use a clock gating scheme with SSI2 whereby all clock signals to SSI2 are disabled until SSI2 is actually being used. This helps to minimize power consumption when audio playback is not being performed. For the MX32, SSI2 is configured in slave mode. The desired framesync and bitclock signals are generated by the MC13783 PMIC.
- The SSI2 transmitter watermark level are also set to support SDMA transfers during audio playback.
- The MC13783 Stereo DAC is then also configured for time-slotted network mode using 4 timeslots/frame and a 44.1 kHz sample rate but operating in slave mode for the i.MX31. For the MX32, the MC13783 Stereo DAC is configured as the master. The first two timeslots are also used to receive the left and right audio channel data words, respectively, to match the SSI2 configuration. If necessary, the required MC13783 audio components are also powered on or reenabled at this time. Normally, the MC13783 audio components that are not actively being used are kept in a power-off or disabled state so as to minimize power consumption.
- The Digital Audio MUX is configured to connect internal port 2 (which is assigned to SSI2) with external port 5 (which is used to communicate with the Stereo DAC). At the same time, the appropriate IO MUX pins are also configured so that the Audio MUX external port 5 signals can actually be routed off-chip to the MC13783.

- The SDMA channel is fully configured to support 16-bit data transfers between the application's memory buffers and the SSI2 TX FIFO0. The SSI2 TX FIFO0 is prefilled with audio data at this point along with the DMA buffers.
- Finally, the SSI2 transmitter is enabled which begins the transmission of the audio data stream.

The hardware repeatedly performs the following functions while audio playback is being performed:

- The SSI will issue a new DMA request whenever the transmitter's FIFO0 level reaches the empty watermark level. The SDMA controller will then refill FIFO0 using data from the DMA buffers until the DMA buffer has been emptied.
- An interrupt is generated whenever a DMA buffer has been emptied and this interrupt is handled by the audio driver. The audio driver is responsible for refilling the DMA buffer and returning it to the SDMA controller for processing.
- Since we are using a double-buffering scheme, the SDMA controller simply uses the other DMA buffer to continue refilling the SSI2 transmitter FIFO0 while the previous DMA buffer is being refilled.

The following hardware changes are made at the completion of each playback operation:

- When we have finished transmitting the entire audio stream, there will be no more data available to refill the empty DMA buffers. Therefore, we can disable the output DMA channel when both output DMA buffers are empty and there is no additional data available to refill them.
- The MC13783 audio components that were used for playback are disabled to minimize power consumption. This step is done before we disable SSI2 to avoid any extraneous noise or "pop" that may be heard over the headphones.
- Finally, we also disable and clock gate SSI2.

3.3.2 Audio Recording

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

- Configure SSI1 for time-slotted network mode using 4 timeslots/frame and a sampling rate of 16 kHz. The first two timeslots are used to transmit the left and right audio channel data words, respectively. Each audio data word is 16 bits long. SSI1 is also configured to operate in master mode using the Serial PLL (for i.MX31) clock signal to generate the appropriate framesync and bitclock signals. Note that we do use a clock gating scheme with SSI1 whereby all clock signals to SSI1 are disabled until SSI1 is actually being used. This helps to minimize power consumption when audio recording is not being performed. For MX32 SSI1 is configured as a slave. The necessary bitclock and framesynch are supplied from MC13783.
- The SSI1 receiver watermark level are also set to support SDMA transfers during audio recording.
- The Digital Audio MUX is configured to connect internal port 1 (which is assigned to SSI1) with external port 4 (which is used to communicate with the Voice CODEC). At the same time, the appropriate IO MUX pins are also configured so that the Audio MUX external port 4 signals can actually be routed off-chip to the MC13783.
- The SDMA channel is fully configured to support 16-bit data transfers between the application's memory buffers and the SSI1 RX FIFO0.

- The SSI1 receiver is enabled and now ready to receive data from the MC13783 Voice CODEC.
- The MC13783 Voice CODEC is then also configured for time-slotted network mode using 4 timeslots/frame and a 16 kHz sample rate but operating in slave mode. For MX32 the MC13783 voice codec is configured in master mode. The first two timeslots are also used to transmit the left and right audio channel data words, respectively, to match the SSI1 configuration. If necessary, the required MC13783 audio components are also powered on or reenabled at this time. Normally, the MC13783 audio components that are not actively being used are kept in a power-off or disabled state so as to minimize power consumption. Note that the Voice CODEC will immediately begin converting the input microphone signal and transmitting audio data to SSI1 once it has been enabled and it receives the correct framesync and bitclock signals .

The hardware repeatedly performs the following functions while audio recording is being performed:

- The SSI will issue a new DMA request whenever the receiver's FIFO0 level reaches the full watermark level. The SDMA controller will then transfer the data from the receiver FIFO0 to an input DMA buffer until the DMA buffer is full.
- The SDMA controller will then generate an interrupt that is handled by the audio driver. The audio driver is responsible for copying the data from the full input DMA buffer into application-supplied buffers and then returning the empty input DMA buffer back to the SDMA controller. Note that any data which cannot be transferred to an application-supplied buffer (for example, due to insufficient space) is simply discarded.
- Since we are using a double-buffering scheme, the SDMA controller simply uses the other DMA buffer to continue recording the data from the SSI1 receiver FIFO0 while the previous DMA buffer is being copied to application-supplied buffers.

The following hardware changes are made at the completion of each recording operation:

- We terminate the recording process by having the application close the audio input stream. At this point, we can disable all of the MC13783 audio components that were used for recording to minimize power consumption.
- Next, we also disable and clock gate SSI1.
- Finally, we disable the input DMA channel to completely terminate the audio recording operation.

3.3.3 Required SoC Peripherals

The audio driver requires the exclusive use of all of the following SoC hardware components:

- Both SSI1 and SSI2 synchronous serial interfaces. By default, SSI1 is used for recording while SSI2 is used for playback.
- The Serial Clock PLL (for i.MX31) to provide the master clock signal for SSI1 and SSI2 (for SSI master mode only). For MX32 the SSI1 and SSI2 are configured in slave mode. The required bitclock and framesync come form MC13783.
- A 14.7 MHz clock signal generator that supplies the CLIA clock input to the MC13783 PMIC (for MC13783 master mode only).
- The Digital Audio MUX to connect both SSI1 and SSI2 to the IO MUX in order to access off-chip peripherals.

- The IO MUX pins for connecting the Digital Audio MUX external ports 4 and 5 to the MC13783 PMIC.
- The SDMA Controller to manage the DMA channels that are used for playback and recording.

3.3.4 Conflicts with Other SoC Peripherals

3.3.4.1 i.MX31/i.MX32 Peripheral Conflicts

There are no conflicts between the SoC peripherals that are required by the audio driver and any other device driver using the default build configurations. However, a potential does exist for peripheral conflicts when developing a customized build configuration. In particular, special care needs to be taken when selecting and configuring the following components:

- The PLL that will be used to provide the main clock signal if the SSI is to be operated in master mode. There are only a limited number of PLLs that are available and the clock signals that they provide may have to be shared with other peripherals. When the SSI is to be operated in master mode, the main clock signal source must be provided at the correct frequency and be properly clock gated so that no peripheral conflicts can occur.
- The IOMUX output pin configuration that is used to route the Digital Audio MUX pins off-chip. A suitable IOMUX configuration must be selected that will allow all of the Digital Audio MUX pins to be routed off-chip without any conflicts with the other peripherals that must also be routed off-chip.

3.3.5 Known Issues

Currently recording is not supported for the MX32ADS due to a number of device driver changes that were implement.to fully optimize the playback operation in terms of maximizing performance and minimzing power consumption.

3.3.6 Required MC13783 PMIC Components

The audio driver requires the exclusive use of all of the following MC13783 PMIC hardware components:

- The Stereo DAC and the audio output section to perform playback.
- The Voice CODEC and the audio input section to perform recording.
- Both digital audio buses in order to transfer data between the SSI and the Stereo DAC and Voice CODEC.
- The CLIA clock input is also required if the Stereo DAC and/or the Voice CODEC are to be operated in master mode.

Note that the audio driver expects that all of the following MC13783 hardware control registers are accessible by the ARM core: RX0, RX1, Audio Codec, Audio Stereo DAC, Audio Tx, and SSI Network. This means that the ARM core must be connected to the MC13783 control registers using the primary processor interface and not the secondary processor interface. This is the normal configuration for all of the currently supported platforms.

3.4 Software Operation

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

Developing a Device Driver → Windows CE Drivers → Audio Drivers → Audio Driver Development Concepts.

3.4.1 Audio Playback

The operation of the audio driver for playback basically follows the hardware configuration steps that were described earlier. Once the appropriate hardware components have been properly configured, then the only thing that the audio driver must still do is to handle the output DMA buffer empty interrupts. This is done via the interrupt handler which simply 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 SDMA controller.

3.4.2 Audio Recording

The operation of the audio driver for recording basically follows the hardware configuration steps that were described earlier. Once the appropriate hardware components have been properly configured, then the only thing that the audio driver must still do is to handle the input DMA buffer full interrupts. This is done via the interrupt handler which simply copies the contents of each input DMA buffer to an application-supplied buffer and then returns the empty DMA buffer to the SDMA controller. If the application-supplied buffer does not have enough space for all of the new data, then any extra data is simply discarded.

The application is signaled using a callback function when the application-supplied buffer is full.

3.4.3 Audio Driver Compile-time Configuration Options

The audio driver can be configured for a wide variety of operating modes depending upon the specific hardware and software requirements. The available compile-time configuration options are described in Table 3-1 and Table 3-2.

The audio driver configuration settings should not be changed without a detailed understanding of the platform's hardware configuration and operating characteristics. Selecting invalid or incorrect configuration settings may result in an audio driver that will not load or work properly. Conversely, the audio driver performance and resource usage can be fine tuned by adjusting these configuration settings. Additional documentation regarding each of the configuration options may be found in the corresponding source files.

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 1.
OUTCHANNELS	Defines the number of output/playback channels that are available. Can be set to either 1 or 2. Default is 2.

Table 3-1. Audio Driver Configuration Options (hwctxt.h)

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.
INSAMPLERATE	The hardware input/recording sampling rate in Hz. Default is 16000.
OUTSAMPLERATE	The hardware output/playback sampling rate in Hz. Default is 44100.
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 INT16.
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 will be clipped to the max/min value by the saturation protection code if USE_MIX_SATURATE is set to 1. Default is 32767 and -32768.
AUDIO_DMA_PAGE_SIZE	The size in bytes of each audio DMA buffer. Default is 6144 bytes.
AUDIO_REGKEY_PREFIX	The common prefix to be used for accessing all of the audio driver runtime configuration registry keys. Default is “Drivers\BuiltIn\Audio\PMIC\Config”.
PLAYBACK_DISABLE_DELAY_MSEC and RECORD_DISABLE_DELAY_MSEC	The delay, in milliseconds, that the audio driver will wait following the completion of an I/O operation before actually disabling the audio CODEC hardware. On some devices, such as the MC13783, there is a significant CODEC warm-up delay before an audio playback or recording operation can be performed. We can delay disabling the audio hardware for a brief period following each audio operation and thereby skip having to re-enable the hardware if another audio I/O operation is started soon after. The delay interval can be set to zero to disable this feature. The default is 1000 for both playback and recording.

Table 3-1. Audio Driver Configuration Options (hwctxt.h)

Configuration Setting Name	Description
BSP_SSI1_MASTER_BOOL and BSP_SSI2_MASTER_BOOL	Selects whether SSI1 and/or SSI2 are to be operated in master mode. Default is TRUE for both settings.For MX32 these are defined as FALSE
SSI1_MASTER_CLOCK_SOURCE and SSI2_MASTER_CLOCK_SOURCE	Defines the master clock input for each SSI. This is only used when the SSI is operating in master mode. The default settings are DDK_CLOCK_BAUD_SOURCE_SERPLL for both settings.
STEREO_DAC_SSI	The SSI that will be used for playback through the Stereo DAC. Default is m_pSSI2.
VOICE_CODEC_SSI	The SSI that will be used for recording using the Voice CODEC. Default is m_pSSI1.
SSI1_AUDMUX_PORT and SSI2_AUDMUX_PORT	The internal Digital Audio MUX ports that are connected to SSI1 and SSI2. The defaults are PORT1 for SSI1 and PORT2 for SSI2.
VOICE_CODEC_AUDMUX_PORT	The external Digital Audio MUX port that is connected to the Voice CODEC. The default is PORT4.
STEREO_DAC_AUDMUX_PORT	The external Digital Audio MUX port that is connected to the Stereo DAC. The default is PORT5.
VOICE_CODEC_AUDIO_BUS	The digital audio bus that connects the Audio MUX to the Voice CODEC. The default is AUDIO_DATA_BUS_1.

Table 3-2. Audio Driver Configuration Options (hwctxt.cpp)

Configuration Setting Name	Description
STEREO_DAC_AUDIO_BUS	The digital audio bus that connects the Audio MUX to the Stereo DAC. The default is AUDIO_DATA_BUS_2.
STEREO_DAC_BUS_MODE	The digital audio bus protocol that is to be used. Either timeslotted NETWORK_MODE or I2S_MODE may be selected. The default is NETWORK_MODE.
VOICE_CODEC_BUS_MODE	The digital audio bus protocol that is to be used. Either timeslotted NETWORK_MODE or I2S_MODE may be selected. The default is NETWORK_MODE.
SSI_SFCSR_TX_WATERMARK and SSI_SFCSR_RX_WATERMARK	The transmitter and receiver watermarks that are to be used with SSI1 and SSI2. The default is 4 for both watermark levels.
DEFAULT_OUTPGA_GAIN	Sets the default output amplifier gain level. The default is OUTPGA_GAIN_MINUS_3DB.

Table 3-2. Audio Driver Configuration Options (hwctxt.cpp)

3.4.4 DMA Support

As indicated above, the audio driver uses the SDMA controller to transfer the digital audio data between the audio application and the SSI FIFOs. This minimizes the processing that is required by the ARM core and can also reduce the power consumption during audio playback and recording operations.

Note, however, that the audio driver always requires the use of DMA support for proper operation. Unlike some of the other device drivers, the audio driver does not have any support for an alternative non-DMA or polling-based operating mode. Therefore, the BSP_SDMA_SUPPORT_SSI1 (for audio recording) and BSP_SDMA_SUPPORT_SSI2 (for audio playback) macros in the bsp_cfg.h header file must always be defined as TRUE even though the audio driver does not explicitly make use of these definitions.

This section will describe the audio driver DMA implementation issues and tradeoffs. The available compile-time DMA-related configuration options will also be described.

In order to use DMA transfers, all of 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 "internal memory" (which is provided by on-chip internal RAM) or "external memory" (which is provided by off-chip external DRAM). Table 3-3 is a summary of the issues and tradeoffs regarding which type of memory should be used for the DMA data buffers.

Table 3-3. 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). Also, less power is required to access the internal RAM than to access
	But the total size of the internal memory region is limited (only 16 kB for the i.MX31).
	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 (128 MB compared to 16 kB for the i.MX31). 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 WinCE system calls.
	The external memory cannot be placed into a low power mode while the DMA is active.

Table 3-4. Configuring for Internal/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. Also set BSP_AUDIO_DMA_BUF_SIZE to the total size (in bytes) for all DMA data buffers that will be allocated.
External	Make sure that the BSP_AUDIO_DMA_BUF_ADDR macro is commented out in bsp_cfg.h

Table 3-4 describes how to configure the build so that the audio driver will allocate its DMA data buffers from either internal or external memory.

The DMA buffer descriptors can also be allocated from either internal or external memory. However, in this case, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). Please refer to the CSPDDK documentation for additional information about the DDKSdmaAllocChain() API.

3.4.5 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 Windows CE 5.0 BSP, the audio module can be disabled, and its clocks turned off, whenever there are no active audio I/O operations. The clock gating as well as the disabling of all related audio hardware components is all handled automatically within the audio module and requires no additional configuration or code changes.

3.4.5.1 PowerUp

This function has been implemented to support resuming an audio I/O operation that was previously terminated by calling the PowerDown() API. We begin by restoring power and re-enabling all of the required audio hardware components. Next we restart the audio DMA transfers to complete the powerup process for the audio driver.

This functionality is currently handled by IOCTL_POWER_SET and theWAV_PowerUp() function is just a stub.In IOCTL_POWER_SET we only check for transition into and out of the suspend state. So we perform a comparison between the current state and the new requested state. If the current power state is D4 (i.e., suspended) and the requested state is anything other than D4 then we call PowerUp() to restore the audio subsystem to the same state that it was in just before PowerDown() was called.

3.4.5.2 PowerDown

The audio driver suspend functionality is currently handled by IOCTL_POWER_SET and the function WAV_PowerDown() is just a stub. In the IOCTL_POWER_SET handler, we first check for the requested power state. If the system is not already in the D4 (suspend) state and the new requested state is D4, then we will call the PowerDown() function to put the audio driver into the suspended state.

Inside PowerDown() we don't need to call Lock() or UnLock() because the PowerDown() thread needs to override and shutdown any audio operation that may currently be in progress. Similarly we shall not resume operation until PowerUp() is called to exit from the suspend state. The PowerDown() function will save the current state of the audio driver and then safely shutdown all of the active audio hardware components, disable all active timers, and terminate both input and output DMA processes. Once the audio driver is in the suspended state, proper operation can only be restored by calling the PowerUp() function.

3.4.5.3 IOCTL_POWER_SET

This Power Manager IOCTL is implemented for the audio driver. All system suspend and resume handling is currently handled by this IOCTL which calls the PowerDown() and PowerUp() functions as described above. For all platforms, the following registry entry must be defined to enable Power Manager support for the audio driver:

```
[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.

3.4.6 Audio Driver Registry Settings

At least one registry key must be properly defined so that the Device Manager will know to load the audio driver when the system is booted. Additional registry keys may also be defined, and even changed at

runtime, to configure the operation of the audio driver. Both the required and optional registry keys for the audio driver are described in the following sections.

3.4.6.1 Required Audio Driver Registry Settings

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

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
          "Prefix"="WAV"
          "Dll"="wavedev_MC13783.dll"
          "Index"=dword:1
          "Order"=dword:10
          "Priority256"=dword:95
          "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}" ; PMCLASS_GENERIC_DEVICE
```

3.4.6.2 Optional Audio Driver Runtime Configuration Registry Settings

The following optional registry keys can also be defined in order to configure the audio driver's various runtime operating modes. If these registry keys are not defined or if the values are invalid, then the values shown below are used as the default settings by the audio driver. Additional configuration settings that are currently supported by the audio driver can be found by looking at the enumerated type definitions in the pmic_audio.h header file. All of the numeric constants that are used in the following registry key values are simply the integer value that corresponds to the enumerated types that are defined in pmic_audio.h for each specific function or item.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\Playback]
   "LeftChannel"=dword:40
   "RightChannel"=dword:80
   "Description"="Stereo headset jack J8 (LeftChannel=0x40, RightChannel=0x80)"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\Recording]
   "LeftChannel"=dword:1
   "RightChannel"=dword:2
   "Description"="Stereo input jack J4 (LeftChannel=1, RightChannel=2)"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\MicBias1]
    "Enable"=dword:0
    "Description"="Microphone bias circuit 1 disabled (0) or enabled (1)"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\MicBias2]
   "Enable"=dword:1
   "Description"="Microphone bias circuit 2 disabled (0) or enabled (1)"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\InputAmp]
    "Mode"=dword:1
    "Mode Description"="Voltage-to-Voltage (1) or Current-to-Voltage (2)"
    "Gain"=dword:8
    "Gain Description"="-8 dB (0) to 23 dB (31 or 0x1F) in 1 dB steps"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio\PMIC\Config\HeadsetDetect]
    "Enable"=dword:0
```

```
"Description"="Disabled (0) or enabled (1)"
```

Note that changes to these audio driver configuration registry keys can be made at any time and the new settings will immediately take effect at the beginning of the next audio I/O operation. A device's current registry entries can be viewed and modified by using the Remote Registry Editor tool that is provide with Platform Builder.

3.5 Unit Test

The audio driver is tested using the Waveform Audio Driver Test suite that is included as part of the Windows CE 5.0 Test Kit (CETK). The test suite includes both automated and interactive tests that are used to test various playback and recording functions.

3.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Stereo headphones or earphones.	This is required to confirm that audio playback is working. The headphones or earphones should have a 3.5mm jack for the MX31 and MX32 platforms.
Mono microphone.	A mono microphone is required in order to perform the audio recording tests. Note that some audio loopback testing can be done by simply putting the microphone near enough to the stereo headphones to record what is being played.

3.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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

3.5.3 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 can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

3.5.4 Running the Audio Driver CETK Tests

The command line for running the audio driver test is ***tux –o –d wavetest***. Alternatively, the CETK GUI interface can also be used from within Platform Builder.

For detailed information about the audio driver tests, see the following section in the Platform Builder online help:

Debugging and Testing → Tools for Debugging and Testing → Windows CE Test Kit → CETK Tests → Waveform Audio Driver Test

The following table describes the audio driver test cases and notes if the tests are expected to pass or fail.

Test Case	Description
100: Build Verification Test	Searches the registry for active audio driver entries and for each audio device entry that is found, the test then tries to select full-duplex operation (that is, simultaneous play and capture). This test should always pass.
1000: Easy Playback	This test plays wave files (e.g., asterisk.wav) from the Windows directory using both the sndPlaySound and PlaySound APIs. You should hear the appropriate sound in the headphones and this test should always pass.
2000: Playback Capabilities	This test reads and displays the audio driver's playback capabilities. The reported audio driver capabilities must be manually confirmed with the features that are supposed to be supported by the audio driver. This test should always pass.
2001: Playback	This test attempts to play a tone using all of the audio formats that are supposed to be supported by the audio driver. This test should always pass.
2002: Playback Notifications	This test attempts to playback a tone using all possible types of callbacks. This test should always pass.
2003: Playback Using Extended Functions	This test attempts to use the extended playback capabilities that are supported by the audio driver such as volume control, playback rate control, and pitch control. This test should always pass.
3000: Capture Capabilities	This test reads and displays the audio driver's recording capabilities. The reported audio driver capabilities must be manually confirmed with the features that are supposed to be supported by the audio driver. This test should always pass.
3001: Capture	This test attempts to record and playback a tone using all of the supported audio formats. This test should always pass.
3002: Capture Notifications	This test attempts to record a tone using all possible types of callbacks. This test should always pass.
4000: Test Volume Control	This test attempts to change the output volume. This test should always pass.

Note that all of the audio recording-related CETK tests will be automatically skipped if audio recording support was disabled in the build configuration.

3.6 System-level Audio Driver Tests

In addition to running the audio driver tests in the CETK, it is also possible to perform various system-level tests that involve the use of the audio driver. The following sections describe various ways to test the audio driver without using the CETK.

3.6.1 Checking for a Boot-time Musical Tune

The normal WinCE boot procedure includes playing a short musical tune just before displaying the touchpanel calibration screen. At this point, the audio driver should already have successfully loaded and you should hear the tune if you attach a headset to the stereo output jack J8 on the MC13783 daughter card.

3.6.2 Confirming Touchpanel Taps and Keypad Key Presses

The normal WinCE system configuration includes the ability to playback a short tapping sound whenever the stylus makes contact with the touchpanel. Again, it is quite easy to confirm whether or not these taps are heard when a headset is attached to the stereo output jack J8 on the MC13783 daughter card.

A similar “click” should also be heard whenever a key on the keypad is pressed.

3.6.3 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 here is that the appropriate software CODECs that may be needed to decode the media file be included in the OS image. The Media Player includes controls for pausing, resuming, and stopping playback as well as advancing it to a specific point. Additional volume and muting controls are also provided.

3.6.4 Using the SDK Sample Audio Applications for Testing

The WinCE SDK that is included as part of 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. Additional information about both the wavrec and wavplay sample applications may be found in the following Platform Builder online help section:

Windows CE Features → Graphics and Multimedia Technologies → Audio → Waveform Audio → Waveform Audio Samples

3.7 Audio Driver API Reference

Detailed reference information for the audio driver may be found in Platform Builder Help at the following location:

Developing a Device Driver → Windows CE Drivers → Audio Drivers → Audio Driver Reference → Waveform Audio Driver Reference

3.8 Audio Driver Troubleshooting Guide

The following sections describe various techniques that may be used to help identify and fix the most common problems involving the audio driver. Note that having access to a digital storage oscilloscope with at least a 20 MHz bandwidth may be very useful when troubleshooting some of the audio-related problems that are described below.

3.8.1 Checking Build-time Configuration Options

Any compile- or link-time errors are probably due to incorrect or invalid configuration settings that were defined in hwctxt.h or hwctxt.cpp. See the Audio Driver Compile-time Configuration Options section above for information about each of the device driver build configuration options.

The build procedure that is documented in the BSP Users Guide must also be followed in order to successfully compile and link the audio driver.

Finally, also confirm that the required Platform Builder catalog items have been included in the OS design. See the Table 1 above for a list of the required and recommended audio driver-related catalog items.

3.8.2 Confirming Audio Driver Loading During Device Boot

First, confirm that the appropriate [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio] registry key has been defined.

Next, confirm that all of the following files exist in the release directory: wavedev_MC13783.dll, waveapi.dll, device.exe. The first DLL is the audio device driver while the second DLL provides the means for applications to access the audio driver. The last file is the Device Manager executable that is required to load the audio driver.

Finally, if you are booting a release image, then you should see all of the following messages in the Platform Builder output window:

```
Loaded symbols for
'D:\WINCE500\PBWORKSPACES\<workspace>\RELDIR\<platform>_ARMV4I_RELEASE\WAVEDEV_MC13783.DLL'
Loaded symbols for
'D:\WINCE500\PBWORKSPACES\<workspace>\RELDIR\<platform>_ARMV4I_RELEASE\WAVEAPI.DLL'
The corresponding messages when booting a debug image are (the timestamp, process ID, and thread
ID numbers may differ from those shown below but the important thing to confirm is that the
modules are being loaded):
4294770428 PID:2bf8a70e TID:2bf9bd56 0x8bf8a4a8: >>> Loading module wavedev_MC13783.dll at
address 0x01A20000-0x01A38000
Loaded symbols for
'D:\WINCE500\PBWORKSPACES\<workspace>\RELDIR\<platform>_ARMV4I_DEBUG\WAVEDEV_MC13783.DLL'
4294770755 PID:2bf8a70e TID:2bf9bd56 0x8bf8a4a8: >>> Loading module waveapi.dll at address
0x03BF0000-0x03C1B000 (RW data at 0x01FCF000-0x01FCF958)
Loaded symbols for
'D:\WINCE500\PBWORKSPACES\<workspace>\RELDIR\<platform>_ARMV4I_DEBUG\WAVEAPI.DLL'
```

3.8.3 Media Player Application Not Found

Make sure that the Media Player catalog item has been included in the OS design (see Table 1 above). The Media Player application will not be included in the final system image if the catalog item is not selected.

3.8.4 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 1 above and make sure that support for the desired audio file format has been included.

3.8.5 No Sound Is Heard During Playback

This assumes that the application is appearing to playback audio normally but nothing is heard in the headset. If the audio application does not appear to be functioning correctly, then please try to use the standard Microsoft-supplied Media Player application first to confirm proper audio playback functionality.

Start by making sure that your headset is plugged into the correct audio jack on the MC13783 daughter card. The default stereo headset output jack is marked J8. You can easily confirm whether or not the output jack is active by plugging and unplugging the headset while the playback operation is underway. You should hear a slight “pop” when the headset is inserted or removed with the output circuit enabled, even if there is no other audio output. This will confirm that the output amplifiers have been enabled and that power is being supplied to the headset. Note that no “pop” will be heard if playback is not currently active, since the audio driver automatically disables all of the audio output circuits when they are not needed in order to minimize power consumption.

Finally, make sure that the volume has not been muted or set to too low a level in the application.

If all of the above checks out, then the next thing to check for is whether or not any digital audio data is actually being transferred. However, an oscilloscope and knowledge about the SSI is required. If an oscilloscope is available, then you should check for the presence of the appropriate signals. As a specific example, the following pins on the EXTEN B connector of the ADS baseboard should be checked (see Figure 3-2 for a sample of the expected signal traces):

1. Pin A12 is the digital audio bus framesync signal. This should be used as the trigger signal and appear as a single bit-wide pulse with a frequency of 44.1 kHz using the default audio driver configuration settings.
2. Pin A9 is the digital audio bus bitclock signal. This signal should be a series of regular pulses with a frequency of 2.8224 MHz using the default audio driver configuration settings.
3. Pin A10 is the digital audio bus TX signal. This signal should show the actual data words for the audio stream. It should be synchronized with the framesync signal on pin A12.

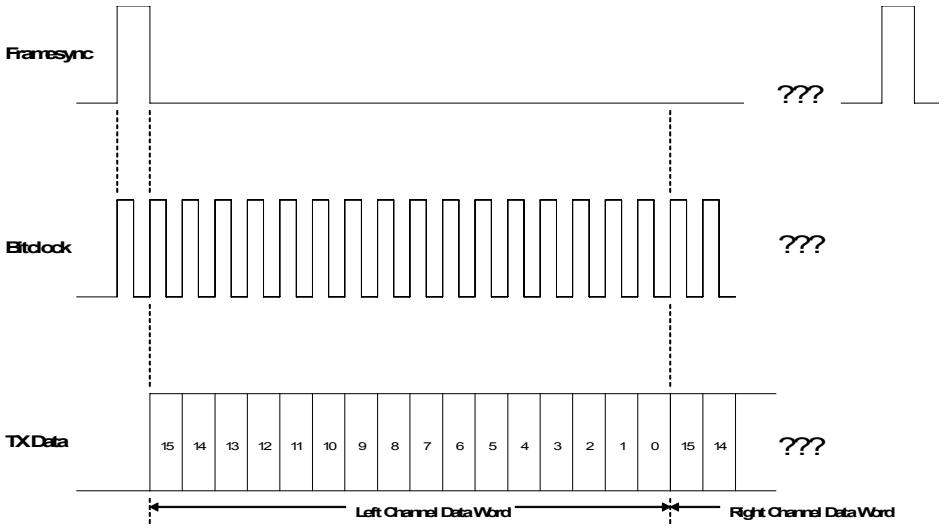


Figure 3-2. Expected Framesync, Bitclock, and Data Signal Traces for Default Audio Driver Configuration

3.8.6 Audio Playback Sounds Distorted, Is Too Fast, or Too Slow

This problem may be caused by any one of the following hardware and/or audio driver configuration errors:

- The PLL is not being configured for the expected frequency to generate the proper SSI framesync and bitclock signals when using SSI master mode. The default audio driver configuration requires that the Serial PLL be configured for a frequency of 220.1472 MHz.
- The audio hardware sampling rate in the audio driver has not been correctly configured. See the description of the OUTSAMPLERATE setting in hwctxt.h in Table 3 above. This setting must correctly match the configuration of the SSI and the PMIC Stereo DAC. Also make sure that the HWSAMPLE and BITSPERSAMPLE definitions are consistent and correct for the hardware platform.

Furthermore, testing audio playback using a debug build image is not recommended because of the additional latency that is introduced by the process of outputting the debug messages. The extra delay caused by having to output a lot of debug messages may cause dropouts and highly distorted audio playback. Building and testing a retail image should avoid this problem.

Chapter 4
Backlight Driver

The backlight drvier has two hardware methods to control the backlight on the LCD display. One way is to use the hardware provied by the display module on the chip and the second is by using the MC13783 Power Management IC (PMIC).

4.1 Smart Backlight Driver

This backlight driver module is used to control the backlight on the smart LCD display. The ADS development board has two hardware methods to accomplish this. One way is to use the Image Processing Unit (IPU) and the second is by using the MC13783 Power Management IC (PMIC). At present, only the PMIC backlight is supported by the driver. It should be noted that the driver controls four LEDs (D4-D7) on the PMIC board, not the circuitry that goes to the LCD backlight. The driver provides 16 levels of control.

4.1.1 Backlight Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
PMIC CSP Driver Path	..\CSP\ARM\FREESCALE\PMIC\MC13783\SDK
CSP Static Library	pmicSdkCsp_MC13783.lib and pmicPdkCsp_MC13783.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\BACKLIGHT\MC13783
Import Library	mxarm11_backlight.lib
Driver DLL	backlight.dll
Catalog Item	Third Party −>BSPs −>Freescale <TGTPLAT> −>Device Drivers −>Backlight −>MC13783 Backlight
SYSGEN Dependency	
BSP Environment Variables	BSP_BACKLIGHT=1 BSP_BACKLIGHT_MC13783=1 BSP_PMIC_MC13783=1 BSP_BACKLIGHT_IPU should not be set.

4.1.2 Requirements

The backlight driver should meet the following requirements:

1. The driver shall conform to the Device Manager streams interface.
2. The driver shall support the <TGTPLAT> MC13783 PMIC hardware method.
3. The driver shall support D4-D7 LEDs on the MC13783 board.

4.1.3 Hardware Operation

The hardware consists of independently programmable current sinking channels. Default behavior for the Backlight Drivers is in Controlled Current Mode, where the drivers are used as programmed current sinks. SPI registers control programmable features such as DC current level, auto ramping / dimming and PWM settings.

Note that Triode Mode current is a function of the supply voltage applied to LEDs. PWM control is retained in Triode Mode, so the average current (and therefore the brightness) of the Backlight LEDs can be pulse width controlled for both the Controlled Current Mode and Triode Mode.

The Triode Mode of operation overrides the Current Control bits when enabled for a given channel. The default POR state is that Triode Mode is disabled. Caution should be taken to ensure that the DC or pulsed current levels are kept within the safe operating area for LEDs since current is not internally controlled in this mode.

4.1.3.1 Conflicts with other SoC peripherals

4.1.3.1.1 i.MX31, i.MX32 Peripheral Conflicts

No conflicts.

4.1.4 Software Operation

The backlight driver is a stream interface driver, and is thus 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. Four possible choices are available for the user.

- IOCTL_POWER_CAPABILITIES - where you 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 – the current power state is returned

4.1.4.1 Backlight Driver Registry Settings

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

[HKEY_CURRENT_USER\ControlPanel\Backlight]

```
"BattBacklightLevel"=dword:7        ; Backlight level settings. 0xF = Full On
"ACBacklightLevel"=dword:F          ; Backlight level settings. 0xF = Full On
"BatteryTimeout"=dword:F
"ACTimeout"=dword:1E
"UseExt"=dword:0                      ; Enable timeout when on external power
"UseBattery"=dword:0                         ; Enable timeout when on battery
```

4.1.5 Unit Test

To test the backlight driver five CETK tests have been written and are available. These are intended to test the physical interface that controls the backlight rather then test the control applet in the control panel. To perform the test a MC13783 board is required. The MC13783 boards have LEDs (D4-D7) available for monitoring the effects of the test.

- Backlight ON Test – turns on the backlight
- Backlight Current Control Mode – places the control hardware into current control mode and runs through a series of timed intervals cycling the duty cycle and pwm values to control the intensity of the backlight hardware
- Backlight Triode Mode – places the control hardware into the triode mode and repeats similar tests as in the current control mode
- Backlight OFF Test – turns off the backlight

Notes:

1. The test cases 11 to 14 of PMICtest.dll described in chapter for Power Management IC (PMIC) are used to do the above tests for testing the PMIC Backlight.

2. Observe the status of LEDs D4 - D7 on the MC13783 card and answer the questions popped up on the LCD screen for testing test cases 11 to 14.

4.1.5.1 Unit Test Hardware

For MX31 and MX32ADS external PMIC card is required. PMIC card is required to monitor D4-D7.

4.1.6 Backlight API Reference

The API for the backlight driver conforms to the stream interface and exposes the standard functions. Further information can be found at **Developing a Device Driver –> Windows CE Drivers –> Streams Interface Drivers**

4.1.6.1 Backlight PDD API Reference

The bulk of the fine backlight control is performed in the platform specific section of the driver. Here the actual hardware interface is controlled via this API.

4.1.6.1.1 PmicBacklightMasterEnable

This function sets the master enable bit of the PMIC backlight and tri-color led controller.

Prototype: PMIC_STATUS PmicBacklightMasterEnable();

Parameters: None

Returns: PMIC_STATUS

4.1.6.1.2 PmicBacklightMasterDisable

This function clears the master enable bit of the PMIC backlight and tri-color led controller.

Prototype: PMIC_STATUS PmicBacklightMasterDisable();

Parameters: None

Returns: PMIC_STATUS

4.1.6.1.3 PmicBacklightRampUp

This function starts backlight brightness ramp up function; ramp time is fixed at 0.5 seconds

Prototype: PMIC_STATUS PmicBacklightRampUp(BACKLIGHT_CHANNEL channel);

Parameters: channel [IN] backlight channel

Returns: PMIC_STATUS

4.1.6.1.4 PmicBacklightRampDown

This function starts backlight brightness ramp down function; ramp time is fixed at 0.5 seconds.

Prototype: PMIC_STATUS PmicBacklightRampDown(BACKLIGHT_CHANNEL channel);

Parameters: channel[IN] backlight channel

Returns: PMIC_STATUS

4.1.6.1.5 PmicBacklightSetMode

This function sets backlight operation mode. There are two modes of operations: current control and triode mode. The Duty Cycle/Cycle Time control is retained in Triode Mode. Audio coupling is not available in Triode Mode.

Prototype: PMIC_STATUS PmicBacklightSetMode(BACKLIGHT_CHANNEL channel, BACKLIGHT_MODE mode);

Parameters: channel[IN] backlight channel

mode[IN] backlight operation mode

Returns: PMIC_STATUS

4.1.6.1.6 PmicBacklightSetCurrentLevel

This function sets backlight current level. In SC55112, LED1 and LED2 are designed for a nominal full scale current of 84mA in 12mA steps. The channels are not individually adjustable, hence the channel parameter is ignored.

level	current
0	0 mA
1	12 mA
2	24 mA
3	36 mA
4	48 mA
5	60 mA
6	72 mA
7	84 mA

Prototype: PMIC_STATUS PmicBacklightSetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8 level);

Parameters: channel[IN] backlight channel

level[IN] current level

Returns: PMIC_STATUS

4.1.6.1.7 PmicBacklightGetCurrentLevel

This function returns the current level for backlight channel

Prototype: PMIC_STATUS PmicBacklightGetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8* level);

Parameters: channel[IN] backlight channel

level[OUT] pointer to current level

Returns: PMIC_STATUS

4.1.6.1.8 PmicBacklightSetDutyCycle

This function sets a backlight channel duty cycle. LED perceived brightness for each zone may be individually set by setting duty cycle. The default setting is for 0% duty cycle; this keeps all zone drivers turned off even after the master enable command. Each LED current sink can be turned on and adjusted for brightness with an independent 4 bit word for a duty cycle ranging from 0% to 100% in approximately 6.7% steps.

Prototype: PMIC_STATUS PmicBacklightSetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8 cycle);

Parameters: channel[IN] backlight channel

cycle[IN] duty cycle

Returns: PMIC_STATUS

4.1.6.1.9 PmicBacklightGetDutyCycle

This function returns the duty cycle for backlight channel

Prototype: PMIC_STATUS PmicBacklightGetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8* cycle);

Parameters: channel[IN] backlight channel

cycle[OUT] duty cycle

Returns: PMIC_STATUS

4.1.6.1.10 PmicBacklightSetCycleTime

This function sets a backlight channel cycle time. Cycle Time is defined as the period of a complete cycle of Time_on + Time_off. The default Cycle Time is set to 0.01 seconds such that the 100 Hz on-off cycling is averaged out by the eye to eliminate flickering. Additionally, the Cycle Time can be programmed to intentionally extend the period of on-off cycles for a visual pulsating or blinking effect.

period	Cycle Time (sec)
0	0.01
1	0.1
2	0.5
3	2

Prototype: PMIC_STATUS PmicBacklightSetCycleTime(UINT8 period);

Parameters: period[IN] cycle time

Returns: PMIC_STATUS

4.1.6.1.11 PmicBacklightGetCycleTime

This function returns the cycle period for backlight controller

Prototype: PMIC_STATUS PmicBacklightGetCycleTime(UINT8* period);

Parameters: period[OUT] pointer to the cycle time

Returns: PMIC_STATUS

4.1.6.1.12 PmicBacklightEnableEdgeSlow

This function enables backlight analog edge slowing mode. Analog Edge Slowing slows down the transient edges to reduce the chance of coupling LED modulation activity into other circuits. Rise and fall times will be targeted for approximately 50usec.

Prototype: PMIC_STATUS PmicBacklightEnableEdgeSlow();

Parameters: None

Returns: PMIC_STATUS

4.1.6.2 PmicBacklightDisableEdgeSlow

This function disables backlight analog edge slowing mode. The backlight drivers will default to an “Instant On” mode.

Prototype: PMIC_STATUS PmicBacklightDisableEdgeSlow();

Parameters: None

Returns: PMIC_STATUS

4.1.7 Power Management

There is no additional power management implementation done specifically for Atlas Backlight other than the implementation described in section 1.5.6 of Power Management IC (PMIC) reference document.

4.2 LCD Backlight Driver

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 should be power-manageable; hence it must meet the requirements of a power-manageable device by implementing the required IOCTLs. The backlight driver will used its own defined timer to set the backlight power states.

4.2.1 Backlight Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	On MX31, MX32: ..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\ BACKLIGHT\DRIVER
CSP Static Library	On MX31, MX32: MXARM11_backlight.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT> \SRC\DRIVERS\BACKLIGHT\DRIVER
Import Library	N/A
Driver DLL	backlight.dll
Catalog Item	Third Party −>BSPs −>Freescale <TGTPLAT> −>Device Drivers −>Backlight −>IPU Backlight
SYSGEN Dependency	
BSP Environment Variables	BSP_BACKLIGHT=1 BSP_BACKLIGHT_IPU=1 BSP_BACKLIGHT_MC13783 should not be set.

4.2.2 Requirements

The backlight driver should meet the following requirements:

1. The driver shall conform to the Device Manager streams interface.
2. The driver shall support the LCDC hardware method.

4.2.3 Hardware Operation

The hardware consists of independently programmable register in LCDC module, PWM Contrast Control Register, which is used to control the signal output at the contrast pin. Default behavior for the Backlight Drivers is to Controls the pulse-width of the built-in pulse-width modulator, which controls the contrast of the LCD screen.

4.2.3.1 Conflicts with other SoC peripherals

No conflicts.

4.2.4 Software Operation

The backlight driver is a stream interface driver, and is thus 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. Four possible choices are available for the user.

- IOCTL_POWER_CAPABILITIES - where you 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 – the current power state is returned

4.2.4.1 Backlight Driver Registry Settings

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:F
    "ACTimeout"=dword:1E
    "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
```

4.2.5 Unit Test

The Backlight driver is tested by Application test.

4.2.5.1 Unit Test Hardware

The following table lists the required hardware to run Backlight Application test.

Requirements	Description
SHARP LQ035Q7DB02 QVGA Panel	Display panel required for display of graphics data.

4.2.5.2 Unit Test Software

The following table lists the required software to run the Backlight application test.

Requirements	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.

4.2.5.3 Running the Backlight Application Test

The following table lists Backlight application test:

Test Cases	Entry Criteria/Procedure/Expected Results
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

4.2.6 Backlight API Reference

The API for the backlight driver conforms to the stream interface and exposes the standard functions. Further information can be found at **Developing a Device Driver → Windows CE Drivers → Streams Interface Drivers**

Chapter 5
Battery Driver

The battery driver module is used to provide information about the battery level to the OS.

The battery driver samples the voltage level upon initialization as well as on power up when coming out of suspend. It then maintains a running count of the capacity that the battery level is at. That capacity of the battery is decremented when the system is powered from the battery and incremented when the system is charging and is powered from the AC adaptor. The rate of discharge is determined by a periodic sampling of the current flowing out of the battery. A similar calculation takes place when the battery is being charged but first a determination is made as to which charging mode is in effect.

5.1 Battery Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
PMIC CSP Driver Path	..\CSP\ARM\FREESCALE\PMIC\MC13783\SDK
CSP Static Library	pmicSdkCsp_MC13783.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\BATTDRVR\MC13783
Import Library	battdrvr_lib.lib
Driver DLL	battdrvr_MC13783.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> MC13783 Battery
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_BATTERY=1, BSP_PMIC_MC13783=1

5.2 Requirements

The battery driver should meet the following requirements:

1. The driver shall conform to the Device Manager streams interface.
2. The driver shall support the <TGTPLAT> MC13783 PMIC.
3. The driver shall support the main battery without the support of the change notification.

5.3 Hardware Operation

The battery driver is implemented with the aid of the MC13783 Power Management Integrated Circuit (PMIC). The PMIC is a multifunctional IC that contains on-chip analog to digital converters used to measure the voltage and current levels of the battery. These levels are then used in determining the capacity level of the battery.

5.3.1 Conflicts with other SoC Peripherals

5.3.1.1 i.MX31, i.MX32 Peripheral Conflicts

No conflicts.

5.4 Software Operation

Upon initialization of the driver the default values for the battery parameters are retrieved from the registry, battery status information is updated .After initialization the function BatteryPDDGetStatus() is called periodically to get the status of the Battery. It 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.

5.4.1 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.
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Battery]
   "Prefix"="BAT"
IF BSP_PMIC_MC13783
   "Dll"="battdrvr_mc13783.dll"
ENDIF
   "Flags"=dword:8                      ; DEVFLAGS_NAKEDENTRIES
   "IClass"="{DD176277-CD34-4980-91EE-67DBEF3D8913}"
   "BattFullLiftTime" = dword:8         ;Batt Spec defined: in unit of hr, here 8hr is assumed
   "BattFullCapacity"=dword:320     ;Batt Spec defined: in unit of mAh, here 800mAhr is assumed
   "BattMaxVoltage"=dword:1068      ;Batt Spec defined: in unit of mV, here 4200mV is assumed
   "BattMinVoltage"=dword:BB8     ;Batt Spec defined: in unit of mV, here 3000mV is assumed
   "BattPeukertNumber"=dword:73     ;Batt Spec defined, here 1.15 is assumed
   "BattChargeEff"=dword:50     ;Batt Spec defined, here 0.80 is assumed
   "PollInterval"=dword:7350; battery polling interval, in milliseconds(30 seconds)

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

5.5 Unit Test

The Battery can be tested, by switching on the system and watching the Power properties window. With time, the charge capacity of the battery can be seen decreasing. Charge capacity decay of the battery is a function of the current flowing out of the battery, the time interval and Peukert number, which is a constant value. So to see the discharging functionality of the battery, the system needs to be booted and the power

properties window has to be observed. Sufficient time should be allowed to get a percentage change in the battery capacity.

Similarly for charging, an external charger of 5V should be connected to the ATLAS charger input. This input is the connector CN6 on the ATLAS card and not the DC input that is supplied to the board. Once a charger voltage greater that 3.795V is detected, the power properties window would change to show a charging status. The gain in capacity would increase gradually and reflect as battery percentage increase over a period of time. Sufficient time should be allowed to see a percentage change in battery capacity.

NOTE

The R34 potentiometer adjusts a voltage level that is monitored by the Power Management IC (PMIC). In order to set the initial capacity of the battery for the test, ensure that the voltage drop across these potentiometers is minimum.

To get a battery capacity of more than 90%, it is best advised that these potentiometers are turned completely in the anticlockwise direction, or in other words set to a minimum value.

5.5.1 Unit Test Hardware

The <TGTPLAT> ADS and the PMIC board are required.

5.6 Battery API Reference

The API for the battery driver conforms to the stream interface and exposes the standard functions. Further information can be found at **Developing a Device Driver –> Windows CE Drivers –> Battery Drivers**

5.6.1 Battery PDD Functions

5.6.1.1 BSPBattdrvrGetParameters

This function returns the battery and charger voltage levels, the current level, and a flag that indicates if the current is charging or discharging.

Prototype

```
BOOL BSPBattdrvrGetParameters(DWORD *pBatt_V, DWORD *pCharger_V, BOOL
*fCharge, DWORD *I)
```

Parameters

pBatt_V

[out] pointer to the battery voltage value

pCharger_V

[out] pointer to the charger voltage value

fCharge

[out] pointer to the flag TRUE for charging FALSE for discharging

I

[out] pointer to the charging/discharging current

5.6.1.2 BSPBattdrvrGetSample

This function returns the battery voltage sample.

Prototype `BOOL BSPBattdrvrGetSample(UINT16 *psample)`

Parameters *psample*

[out] pointer to the sample value,
psample[0] = battery voltage;
psample[1] = battery current;
psample[2] = charger voltage;
psample[3] = charger current;

5.6.2 Battery Driver Structures

5.6.2.1 Battery Channels Structure

```
typedef enum _BATTDRVR_CHANNELS {
    BattVoltage,
    BattCurrent,
    ChargerVoltage,
    ChargerCurrent,
    TotalChannels,
} BATTDRVR_CHANNELS;
```

5.6.2.2 Battery Information Structure

```
typedef struct _BATT_INFO
{
    DWORD        adc_level;
    DWORD        adc_batt_max_V;
    DWORD        adc_batt_min_V;
    DWORD        adc_batt_max_I;
    DWORD        adc_batt_min_I;
    DWORD        adc_charger_max_V;
    DWORD        adc_charger_min_V;
    DWORD        adc_charger_max_I;
    DWORD        adc_charger_min_I;
    DWORD        charger_V_limit;
} BATT_INFO, *PBATT_INFO;
```

5.7 Power Management

There is no additional power management implementation done specifically for Battery driver other than the implementation described in section 1.5.6 of Power Management IC (PMIC) reference document.

Chapter 6
Boot from NAND and SD

6.1 Introduction

Boot support from NAND or SD/MMC includes the following

- Xloader
- Secondary Program Loader (will be referred as bootloader in this document)
- Storing OS binary image

6.2 Xloader

6.2.1 Xloader for NAND Flash

6.2.1.1 NAND Flash XLDR summary

Xloader is a primary boot loader whose responsibility is to copy the boot loader from the NAND flash and then pass the execution to boot loader. Since we have 2KB of NFC buffer, therefore we have a tiny sized (maximum size being 2KB) primary boot loader i.e. Xloader whose job is to bring up the boot loader which has the main boot loader functionality.

Xloader relies totally on Boot ROM code to initialize the NAND flash controller which in turn initializes the device, and then the NFC controller copies first 2KB data of Block 0 to NFC buffer.

Currently, NAND Flash Xloader has support for following NAND devices

- K9K1G08U0B - Samsung SLC NAND with 512 Bytes page
- K9F1G08U0A - Samsung SLC NAND with 2K page
- K9LAG08U0M - Samsung MLC NAND with 2K page

NOTE

- There is no support for MLC NAND in MX31.
- Maximum size of NAND XLDR is 2K.

The following table provides a summary of source code location, library dependencies and other BSP information

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A

CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\BOOTLOADER\XLDR\NAND
Import Library	N/A
Module Name	xldr
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NAND_FMD=1 & BSP_NAND_K9K1G08U0B=1 for Samsung SLC NAND with 512 Bytes page BSP_NAND_K9F1G08U0A=1 for Samsung SLC NAND with 2K page BSP_NAND_K9LAG08U0M=1 for Samsung MLC NAND with 2K page

6.2.1.2 Implementation

Xloader is the first code to execute on system while booting from external NAND flash memory. It takes the responsibility of loading the boot loader for continuing boot procedure. Xloader is stored in first 2K bytes of block 0 of the NAND device. This allows the NAND Xloader image to be updated independently by the bootloader.

While booting from NAND flash, Boot ROM code copies Xloader in the NFC buffer available and assigns the program counter to the base of the NFC buffer. Xloader starts executing from the base address of this buffer and does the basic initialization of the processor along with SDRAM initialization. Since, NFC buffer is a single port RAM, we cannot copy boot loader to external RAM while executing from NFC buffer. Hence, after SDRAM initialization, Xloader does a self copy of 2KB data copied in NFC buffer to external RAM and then starts executing from external RAM and copies the boot loader for continuing the boot procedure which in turn brings up the OS.

6.2.2 Xloader for SD/MMC

6.2.2.1 SD/MMC XLDR Summary

Xloader is a primary boot loader whose responsibility is to copy the boot loader from the SD/MMC memory and then pass the execution to boot loader. Since MX32 Internal RAM is limited to 16KB, the boot loader's memory is not enough to occupy and execute within the 16KB Internal RAM. Therefore we have a tiny sized primary boot loader i.e. Xloader whose job is to bring up the boot loader which has the main boot loader functionality.

Xloader relies totally on Boot ROM to initialize the SD/MMC memory and bring it to a proper state.

Currently, SD/MMC Xloader has support only for following cards

- High (greater than or equal to 2 GB) and low capacity (less than 2 GB) MMC cards
- Low capacity (less than 2 GB) SD cards. The limitation for High Capacity SD cards is because Boot ROM doesn't support it.

Currently, SD/MMC has support for following devices

- SD cards - SanDisk (256MB, 512MB), Transcend (128MB, 512MB)

- MMC cards - Transcend (128MB, 1GB), Silicon Motion (2GB, 4GB)

NOTE

- Only SDHC1 controller supports SD/MMC boot
- SD/MMC XLDR occupies a maximum size of 4K.

The following table provides a summary of source code location, library dependencies and other BSP information

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX32ADS
Target SOC (TGTSOC)	MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\BOOTLOADER\XLDR\SD
Import Library	N/A
Module Name	xldr
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_LC_MMC=1 for low capacity SD/MMC cards BSP_LC_MMC=0 for high capacity MMC cards only

6.2.2.2 Implementation

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. It configures the memory only in 1-bit mode and brings it to Transfer state where read/write operation can be done on the memory. The Boot ROM then copies first 4KB memory location of the SD/MMC memory to Internal RAM and passes the control to the Xloader. The Xloader in turns initializes the SDRAM, copies the boot loader from a predefined memory location of the SD/MMC memory to SDRAM and pass control to boot loader which in turn brings up the OS. Xloader reads data in 1-bit mode only.

SD/MMC boot supports both secure as well as insecure boot. The memory layout of Xloader has been modified to easily the hook the security features with the Xloader code. Please refer Section 6.3.3, SD/MMC Xloader Layout in Internal RAM. The first instruction in the memory layout is jump to the Xloader entry code. The first 1KB after the jump instruction is for the security library which will be used by Xloader for security verification. The security certificates are stored at the lower 1KB of the 4KB memory layout.

6.2.3 Boot Loader for NAND Flash

6.2.3.1 NAND Flash Boot Loader Summary

Currently, NAND Flash Boot Loader has support for following NAND device

- K9K1G08U0B - Samsung SLC NAND with 512 Bytes page
- K9F1G08U0A - Samsung SLC NAND with 2K page
- K9LAG08U0M - Samsung MLC NAND with 2K page

NOTE

There is no support for MLC NAND in MX31

The following table provides a summary of source code location, library dependencies and other BSP information

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\BOOTLOADER\ ..\PLATFORM\<*TGTPLAT*>\SRC\COMMON\NANDFMD
Import Library	N/A
Module Name	eboot/sboot
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NAND_FMD=1 & BSP_NAND_K9K1G08U0B=1 for Samsung SLC NAND with 512 Bytes page BSP_NAND_K9F1G08U0A=1 for Samsung SLC NAND with 2K page BSP_NAND_K9LAG08U0M=1 for Samsung MLC NAND with 2K page

6.2.3.2 Implementation

NAND bootloader implements the following features:

- Support for storing Boot Loader and NAND Xloader images to NAND Flash.
- Support for storing OS images to NAND flash
- Support for loading OS image from NAND flash to RAM.

In order to achieve following features, FMD (Flash Media Driver) for NAND device is used which exposes functions to perform erase, read and write operations on NAND flash.

Boot loader, Xloader and OS region is protected by marking the ‘bOEMReserved’ byte of the metadata area as ‘0x00’ for each sectors corresponding to these regions. This prevents corruption of these regions

by flash abstraction layer after the OS comes up. Boot loader also takes care of identifying and skipping bad blocks including ones marked by the NAND device manufacturer.

For preparing and downloading NAND boot loader, please refer to section “Preparing for Downloading and Debugging” in the BSP users guide.

For usage of the NAND boot loader, please refer to section “Downloading and Debugging Images” in the BSP users guide.

Please refer Section 6.3.1, NAND Flash Memory Layout for the break up of regions within the nand flash device.

6.2.4 Boot Loader for SD/MMC

6.2.4.1 SD/MMC Boot Loader Summary

The following table provides a summary of source code location, library dependencies and other BSP information

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX32ADS
Target SOC (TGTSOC)	MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\BOOTLOADER\ ..\PLATFORM\<*TGTPLAT*>\SRC\COMMON\SDFMD
Import Library	N/A
Module Name	eboot/sboot
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	N/A

6.2.4.2 Implementation

SD/MMC boot implements the following features:

- Support for storing Boot Loader and SD/MMC Xloader images into SD/MMC flash.
- Support for storing OS images to SD/MMC flash
- Support for loading OS image from SD/MMC flash to RAM.

In order to achieve following features, FMD (Flash Media Driver) for SD/MMC device is used which exposes functions to perform erase, read and write operations on SD/MMC flash. The FMD layer provides support for all types of cards high as well low capacity SD and MMC cards. It also supports for 1 and 4 bit modes data transfer.

For preparing and downloading SD/MMC boot loader, please refer to section “Preparing for Downloading and Debugging” in the BSP users guide.

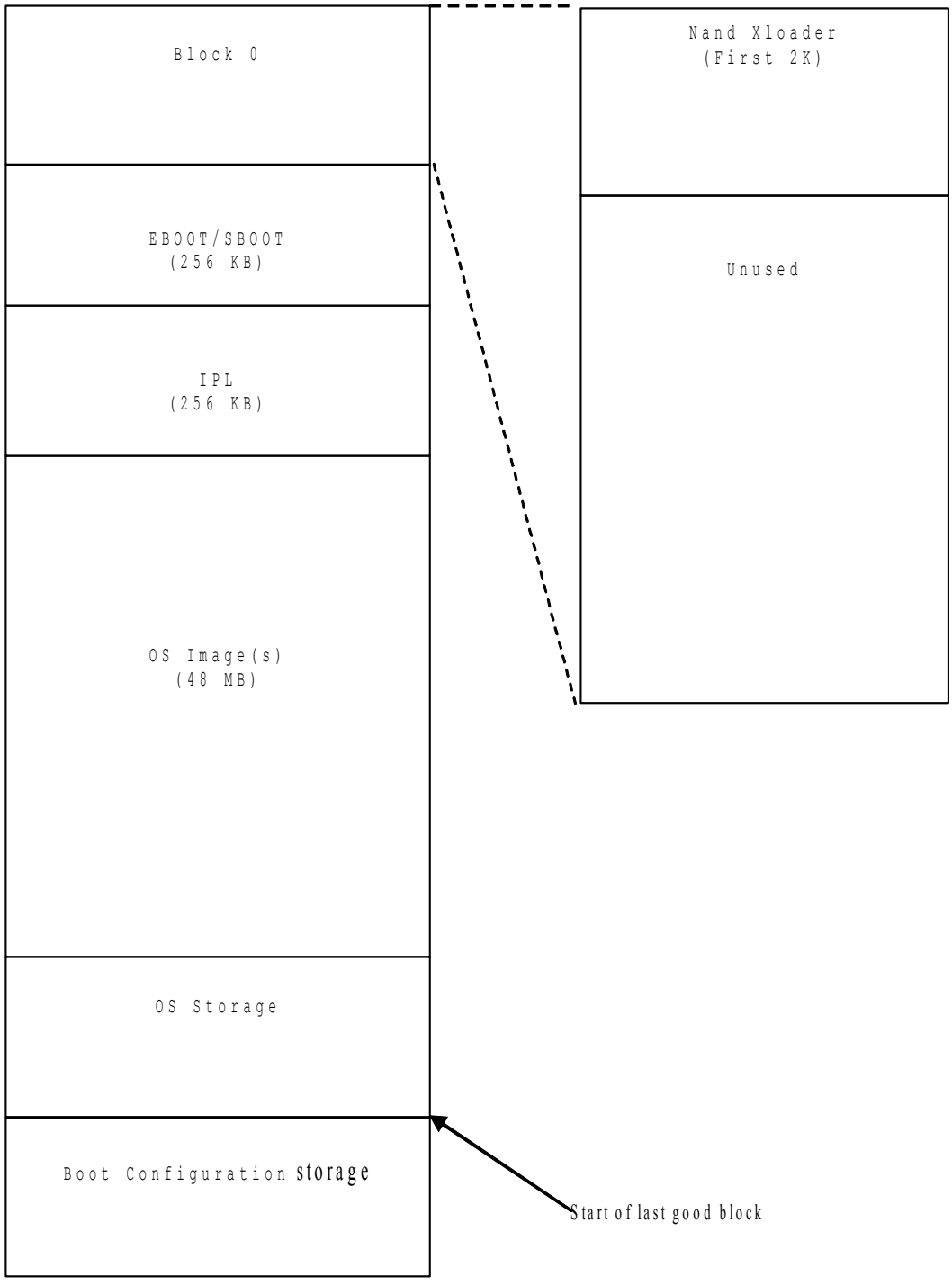
For usage of the SD/MMC boot loader, please refer to section “Downloading and Debugging Images” in the BSP users guide.

Please refer Section 6.3.2, SD/MMC flash Memory Layout for the break up of regions within the SD/MMC memory.

6.3 Memory Layout

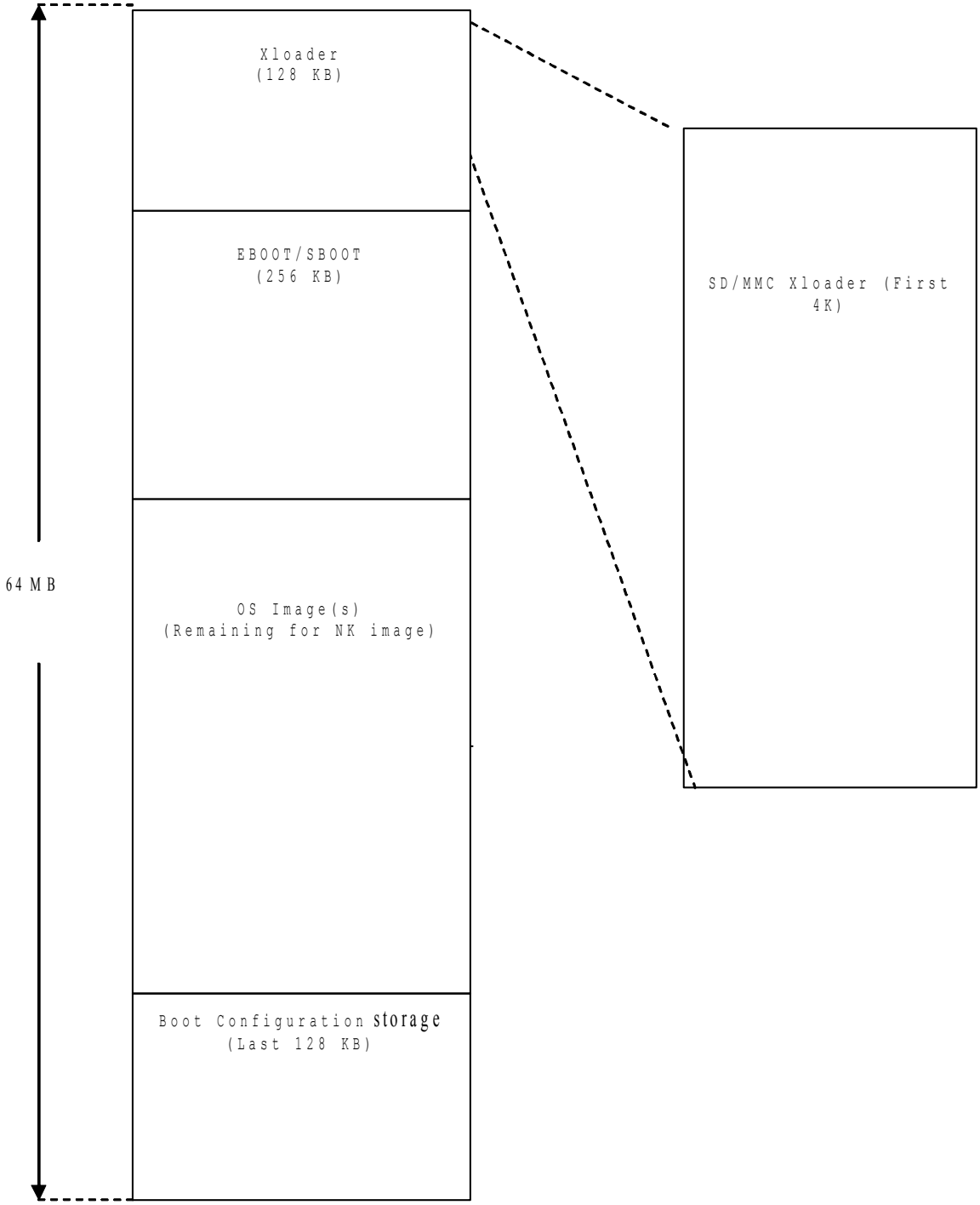
6.3.1 NAND Flash Memory Layout

The diagram below depicts the NAND flash allocation used in the MX32 BSP



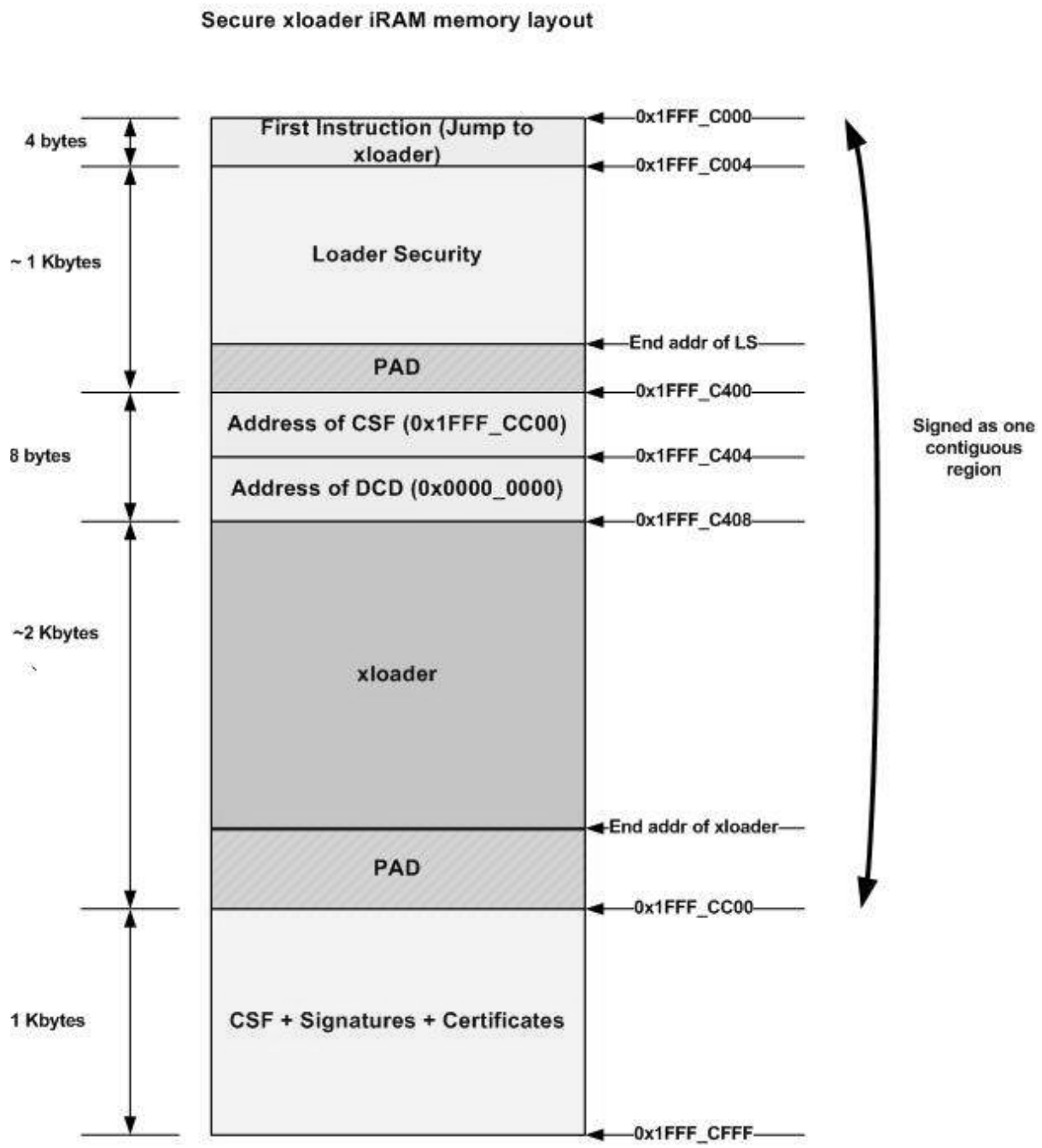
6.3.2 SD/MMC flash Memory Layout

The diagram below depicts the SD/MMC flash allocation used in the MX32 BSP



6.3.3 SD/MMC Xloader Layout in Internal RAM

The diagram below depicts the Xloader layout in Internal RAM used in the MX32 BSP



Chapter 7
Camera Driver

7.1 Camera Driver Summary

The Camera Driver is based on the Windows CE Video Camera Device Driver Interface, introduced with Windows Mobile 5.0. This interface provides basic support for video and still image capture devices. The camera driver conforms to the architecture for Windows CE stream interface drivers, and allows applications to use the middleware layer provided by the DirectShow video capture infrastructure to communicate with and control the camera. This module is designed to be compatible with the iMagic IM88023B1 and Magna 521DA camera sensor modules.

At the lower layer, the Camera Driver performs several tasks, configuring and controlling the following hardware components: camera sensors, image capture hardware, hardware for pre-processing captured camera images, and display hardware for the direct display of video preview data.

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\IPU\CAMERA
CSP Driver Path	N/A
CSP Static Library	mxarm11_camera.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\IPU\CAMERA
Import Library	N/A
Driver DLL	camera.dll
Catalog Items	N/A
SYSGEN Dependency	SYSGEN_DSHOW_CAPTURE=1
BSP Environment Variables	BSP_CAMERA=1 IMGCAMERAOEM=1

7.2 Requirements

The Camera driver should meet the following requirements:

1. The driver shall support the Windows CE Video Camera Device Driver Interface.

2. The driver shall support Preview, Capture, and Still pins.
3. The driver shall support a Direct-to-Display preview mode.
4. The driver shall support the iMagic IM88023B1 and the Magna 521DA camera sensors.
5. The driver shall support up to a 30 frames per second rate at VGA resolution.
6. The driver shall support two power management modes, full on and full off.

7.3 Hardware Operation

7.3.1 For MX31 and MX32

Several hardware modules are involved in the operation of the Camera driver. An iMagic IM88023B1 or Magna 521DA camera sensor, configured through the I2C interface, captures external image data. All other hardware elements of the Camera driver are within the Image Processing Unit (IPU). The IPU Camera Sensor Interface (CSI) receives data from the sensor and converts the data into a format understood by the IPU. This data subsequently flows through the IPU Image Converter (IC) module, where it undergoes pre-processing. There are two pre-processing paths: one for encoding and one for viewfinding. The pre-processed image data is then transferred by the IPU DMA module to one of two destinations: system memory (encoding or viewfinding data) or the IPU Synchronous Display Controller (SDC) for display (viewfinding data).

For detailed operation and programming information, refer to the chapter on the Image Processing Unit (IPU) in the hardware specification document.

7.3.2 Conflicts with other SoC peripherals

7.3.2.1 MX31/MX32 Peripheral Conflicts

There are no peripheral conflicts on this SoC.

7.4 Software Operation

7.4.1 Communicating with the Camera

Communication with the camera driver is accomplished through Camera APIs defined by Microsoft for Windows Mobile 5.0. Applications may access these Camera APIs directly or through the DirectShow video capture support.

7.4.1.1 Using the Windows CE Video Camera Device Driver Interface

The Windows CE Video Camera Device Driver Interface provides basic support for video and still image capture devices. Please refer to the following Windows Mobile 5.0 Help Documentation section for information on using these Camera APIs: **Developing a Device Driver –> Windows Mobile-based Device Drivers –> Camera Drivers –> Camera Driver Reference.**

7.4.1.2 Using DirectShow for Video Capture

DirectShow provides support in its architecture for the creation of filter graphs for video capture. Information on using DirectShow for video capture can be found in the following Windows Mobile 5.0 Help Documentation section: **Windows Mobile-based Device Features –> Graphics and Multimedia Technologies –> Media –> DirectShow –> DirectShow Application Development –> Audio and Video Capture Support –> Video Capture.**

7.4.2 Camera Registry Settings

Two sets of registry settings are important for proper Camera Driver operation. One set is for the camera sensor, and the other is for the Camera Driver.

The following registry keys are required to properly select the camera sensor used on the SoC.

```
[HKEY_LOCAL_MACHINE\Drivers\CSI]
   "CameraId"=dword:1
```

The CameraId registry key selects between the available camera sensor modules. The valid values are the following:

- 1 to indicate that the camera sensor in use is the iMagic IM88023B1.
- 2 to indicate that the camera sensor in use is the Magna 521DA.

The following registry keys are required to properly load the Camera Driver.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CAMERA]
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1
   "IClass"="{CB998A05-122C-4166-846A-933E4D7E3C86}"

[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture]
   "Prefix"="PIN"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1
   "IClass"=multi_sz:"{C9D092D6-827A-45E2-8144-DE1982BFC3A8}",
                     "{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

7.4.3 Power Management

The camera driver consumes power primarily through the operation of various IPU sub-modules, such as the CSI, which synchronizes and receives image data from the camera sensor, and the IC, which performs pre-processing operations on captured image data. The CSI and IC modules are enabled when the camera is set to a running state. At all times when the camera is not running, the CSI and IC modules are disabled.

Support for transitioning to the Suspend and Resume states if provided through the IOCTL_POWER_SET IOCTL.

7.4.3.1 PowerUp

This function is not implemented for the camera driver.

7.4.3.2 PowerDown

This function is not implemented for the camera driver.

7.4.3.3 IOCTL_POWER_SET

The camera 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 – Action is only taken when resuming from the D4 state. If the camera was running when the transition to the D4 state occured, the camera returns to a running state, re-enabling the CSI and IC modules.
- D4 – Action is only taken if the camera is running when the request to transition to the D4 state occurs. In this case, the camera is stopped and the CSI and IC modules are disabled.

7.5 Unit Test For MX31/MX32

Since the Camera Driver API was introduced with Windows Mobile 5.0, there are no camera tests provided by Microsoft for Windows CE 5.0. Thus, on Windows CE 5.0, a custom test and a custom application are provided to test the Camera Driver.

7.5.1 Unit Test Hardware

The following table lists the required hardware to run the Windows CE 5.0 custom Camera CETK test and the Windows CE 5.0 custom Camera application.

Requirements	Description
iMagic IM88023B1 camera sensor or Magna 521DA camera sensor	Camera sensor required for capture of camera image data.

7.5.2 Unit Test Software

7.5.2.1 Custom Camera CETK Test

The following table lists the required software to run the custom Camera Test.

Requirements	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

Requirements	Description
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
Camtest.dll	Main test .dll file.

7.5.2.2 Custom Camera Application

The following table lists the required software to run the custom Camera Application.

Requirements	Description
Camapp.exe	Executable file for the camera application.

7.5.3 Building the Camera Tests

7.5.3.1 Custom Camera CETK Test

In order to build the custom Camera test, complete the following steps:

Build an OS image for the desired configuration

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the Camera Test directory. (for MX31/MX32, \WINCE500\SUPPORT\TESTS\Camera)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the camtest.dll file will be located in the $(_FLATRELEASEDIR) directory.

7.5.3.2 Custom Camera Application

In order to build the custom Camera Application, complete the following steps:

Build an OS image for the desired configuration

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the Camera Application directory. (for MX31/MX32, \WINCE500\SUPPORT\APPS\CAMAPP)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the camapp.exe file will be located in the $(_FLATRELEASEDIR) directory.

7.5.4 Running the Camera Tests

7.5.4.1 Running the Custom Camera CETK Test

The following command executes the Custom Camera CETK Test:

tux –o –d camtest.dll

The following table describes the test cases contained in the test suite:

Test Case	Description
1	This function initializes the camera for testing. The following tasks are performed: - Retrieving panel dimensions, so that we can draw directly to the framebuffer during testing. - Creating message queues for each pin instance - Creating a pin instance for PREVIEW, CAPTURE, and STILL. - Reading the Datarange information for each pin into a global variable.
2	This function tests the functionality of camera capture. Images are captured for the CAPTURE Pin. It contains a loop in which a CAPTURE pin image is read from its message queue, drawn the display, and then a new buffer in enqueued. Intermittently, the camera driver is modified. These modifications include changing the camera zoom and modifying the flipping orientation.
3	This function test the functionality of camera preview. Images are capture for the PREVIEW Pin. It contains a loop in which a PREVIEW pin image is read from its message queue, and then a new buffer is enqueued. Intermittently, the PREVIEW pin or the camera driver is modified. These modifications include changing the camera zoom, modifying the flipping orientation, and changing the display offset in the display panel. Additionally, the STILL pin is tested, capturing a still image and displaying it to the display panel.

7.5.4.2 Running the Custom Camera Application

The following command executes the Custom Camera Application:

camapp.exe

7.6 Camera Driver API Reference

Documentation for the camera driver APIs can be found within the latest Windows Mobile Version 5.0 Documentation. There is one additional custom API provided to allow applications to enable direct display of video preview data.

Reference information on basic camera driver functions, methods, and structures can be found at the following location in the Windows Mobile documentation:

Developing a Device Driver –> Windows Mobile-based Device Drivers –> Camera Drivers –> Camera Driver Reference

7.6.1 IOCTL_SET_DIRECT_DISPLAY_MODE

This Camera **DeviceIoControl** request turns Direct Display mode on or off. With Direct Display mode on, when the PREVIEW pin is set to the RUN state, the video preview data will be sent directly to the display.

Parameters *nInBufferSize*

TRUE (or 1) to enable Direct Display mode, FALSE (or 0) to disable Direct Display mode.

Chapter 8
Chip Support Package Driver Development Kit (ARM11 CSPDDK)

The BSP includes a component called the Chip Support Package Driver Development Kit (CSPDDK) which 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:

- System clocking (CCM)
- GPIO
- DMA (SDMA)
- Pin multiplexing and pad configuration (IOMUX)

8.1 CSPDDK Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\CSPDDK
<TGTPLAT> CSP Driver Path	..\CSP\ARM\FREESCALE\<TGTPLAT>\DRIVERS\CSPDDK
CSP Static Library	<*TGTSOC*>_ddk.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\CSPDDK
Import Library	cspddk.lib
Driver DLL	cspddk.dll
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	Remove BSP_NOCSPDDK=1

8.2 Requirements

The CSPDDK driver should meet the following requirements:

1. The driver shall support an interface that allows synchronized inter-process access to the following set of shared SoC resources:
   — GPIO (DDK_GPIO)

- SDMA (DDK_SDMA)
- IOMUX (DDK_IOMUX)
- CCM (DDK_CLK)

2. The driver shall expose exported functions that can be invoked without incurring a system call (i.e. not a stream driver)

8.3 Hardware Operation

Refer to the hardware specification document for detailed operation and programming information.

8.3.1 Conflicts with other SoC peripherals

8.3.1.1 i.MX31 Peripheral Conflicts

Refer to the i.MX31 hardware specification document for possible conflicts.

8.3.1.2 i.MX32 Peripheral Conflicts

Refer to the i.MX32 hardware specification document for possible conflicts.

8.4 Software Operation

8.4.1 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.

8.4.2 Compile-Time Configuration Options

The CSPDDK exposes compile-time options for configuring the SDMA support. In some cases, these compilation variables are also leveraged by driver code to expose a central point of controlling SDMA functionality. The following table describes the available CSPDDK compile options:

Compilation Variable	Header Location	Description
IMAGE_WINCE_DDKSDMA_IRAM_PA_START	image_cfg.h	Physical starting address in internal RAM (IRAM) where the shared SDMA data structures will be located. **Note:** Only applicable to MX32ADS.

IMAGE_WINCE_DDKSDMA_IRAM_OFFSET	image_cfg.h	Offset in bytes from the base of IRAM for the SDMA data structures. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKSDMA_IRAM_SIZE	image_cfg.h	Size in bytes of the IRAM reserved for SDMA data structures. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_CSPDDK_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared CSPDDK data structures will be located. The DDK_CLK and DDK_SDMA will use space from this region. This address must correspond to the region reserved in config.bib. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_CSPDDK_RAM_PA_OFFSET	image_cfg.h	Offset in bytes from the base of external RAM for the shared CSPDDK data structures. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_CSPDDK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for CSPDDK data structures. This size must correspond to the region reserved in config.bib. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKSDMA_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_SDMA data structures will be located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKSDMA_RAM_PA_OFFSET	image_cfg.h	Offset in bytes from the base of external RAM for the shared DDK_SDMA data structures. This offset must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKSDMA_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_SDMA data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKCLK_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_CLK data structures will be located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.

IMAGE_WINCE_DDKCLK_RAM_PA_OFFSET	image_cfg.h	Offset in bytes from the base of external RAM for the shared DDK_CLK data structures. This offset must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.
IMAGE_WINCE_DDKCLK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_CLK data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions. **Note:** Only applicable to MX32ADS.
IMAGE_SHARE_IRAM_SDMA_PA_START	image_cfg.h	Physical starting address in internal RAM (IRAM) where the shared SDMA data structures will be located. **Note:** Only applicable to MX31.
IMAGE_SHARE_IRAM_SDMA_OFFSET	image_cfg.h	Offset in bytes from the base of IRAM for the SDMA data structures. **Note:** Only applicable to MX31.
IMAGE_SHARE_IRAM_SDMA_SIZE	image_cfg.h	Size in bytes of the IRAM reserved for SDMA data structures. **Note:** Only applicable to MX31.
IMAGE_SHARE_SDMA_PA_START	image_cfg.h	Physical starting address in external RAM where the shared SDMA data structures will be located. This address must correspond to the region reserved in config.bib. **Note:** Only applicable to MX31.
IMAGE_SHARE_SDMA_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for SDMA data structures. This size must correspond to the region reserved in config.bib. **Note:** Only applicable to MX31.
BSP_SDMA_MC0PTR	bsp_cfg.h	Starting address for the shared SDMA data structures. For MX31, set to IMAGE_SHARE_IRAM_SDMA_PA_START to use internal RAM or IMAGE_SHARE_SDMA_PA_START to use external RAM. For MX32ADS, set to IMAGE_WINCE_DDKSDMA_IRAM_PA_START to use internal RAM or to IMAGE_WINCE_DDKSDMA_RAM_PA_START to use external RAM.

BSP_SDMA_CHNPRI_xxx	bsp_cfg.h	Assigns a SDMA channel priority to the respective peripheral. Refer to the individual driver chapters for more information on the specific priorities.
BSP_SDMA_SUPPORT_xxx	bsp_cfg.h	Boolean to specifies if SDMA-based transfers are enabled for each respective peripheral. Refer to the individual driver chapters for more information on the DMA support provided.

The CSPDDK will manage the allocation of buffer descriptor chains for drivers and applications. The allocation scheme will first attempt to allocate the buffer descriptor chain from a fixed memory pool within the region specified by BSP_SDMA_MC0PTR. If the CSPDDK is unable to allocate enough storage from this fixed pool, it will dynamically allocate the necessary storage from external memory.

To decrease power consumption in system uses cases such as audio playback, it is beneficial to configure BSP_SDMA_MC0PTR to point to a reserved internal RAM (IRAM) region and allocate the audio buffers in IRAM. This configuration does not require external memory cycles in the data flow from the audio buffers to the SSI and allows the CSPDDK to utilize EMI clock gating to significantly reduce the power consumption. Refer to the audio chapter in the Reference Guide for more information on configuring audio DMA support.

8.4.3 Registry Settings

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

8.4.4 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/PowerDown stream interface. However, the CSPDDK functions will be invoked by other drivers during power state transitions.

8.5 Unit Test

Due to the heavy use of the CSPDDK routines by other drivers on the system, the CSPDDK tests are currently limited to testing the interface exposed by the DDK_SDMA.

8.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
No additional hardware required.	

8.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
SDMATEST.dll	Test .dll file

8.5.3 Building the Unit Tests

In order to build the CSPDDK tests, complete the following steps:

Build an OS image for the desired configuration:

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the SDMA Tests directory. (\WINCE500\SUPPORT\TESTS\SDMA)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the SDMATEST.dll file will be located in the $(_FLATRELEASEDIR) directory.

8.5.4 Running the Unit Tests

The command line for running the DDK_SDMA tests is ***tux –o –d SDMATEST***. The CSPDDK_SDMA tests do not contain any test-specific command line options.

The following table describes the test cases contained in the DDK_SDMA tests.

Test Case	Description
1: SDMA Open/Close Channel	Tests open/close operation of the SDMA virtual channels. Attempts to open all available channels and verify that the correct virtual channel ID is returned. All successfully opened channels are then closed.
2: SDMA Memory-to-Memory	Tests the SDMA's ability to perform a memory-to-memory transfer. A virtual channel is requested and then DMA buffers are used to define a memory transfer. The transfer is done in both directions and the results are verified. This transfer is interrupt-driven and uses the standard OAL interrupt registration procedures normally used by device drivers.

8.6 CSPDDK DLL Reference

8.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

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

8.6.1.1 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_CLOCK_CKO_SRC	Clock output source (CKO) signal selections.
DDK_CLOCK_CKO_DIV	Clock output source (CKO) divider selections.

8.6.1.2 DDK_CLK Functions

8.6.1.2.1 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.

8.6.1.2.2 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.

8.6.1.2.3 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.

8.6.1.2.4 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.

8.6.1.2.5 DDKClockSetCKO

This function configures the clock output source (CKO) signal.

```
BOOL DDKClockSetCKO(
        BOOL bEnable,
        DDK_CLOCK_CKO_SRC src,
        DDK_CLOCK_CKO_DIV div)
```

Parameters

bEnable [in] Set to TRUE to enable CKO output. Set to FALSE to disable CKO output.

src [in] Selects the CKO source signal.

div [in] Specifies the CKO divide factor.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.1.2.6 DDKClockEnablePanicMode

This function informs the DVFC driver that a panic condition exists and forces the system to immediately transition to the highest DVFS setting. This function can be used to prevent the DVFS logic from requesting a lower frequency and voltage setting.

NOTE

This function is not available on the MX32ADS platform.

```
BOOL DDKClockEnablePanicMode(void)
```

Parameters None.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.1.2.7 DDKClockDisablePanicMode

This function informs the DVFC driver that a panic condition no longer exists and allows the system to return to normal DVFS operation.

NOTE

This function is not available on the MX32ADS platform.

```
BOOL DDKClockDisablePanicMode(
        UINT32 ratio)
```

Parameters

ratio [in] Specifies the integer ration used to scale the AHB bus clock.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.1.2.8 DDKClockBusScale

This function requests the DVFC driver to scale the AHB bus clock by an integer ratio.

WARNING

This function cannot be used on platforms using DDR memory.

NOTE

This function is not available on the MX32ADS platform.

```
BOOL DDKClockBusScale(void)
```

Parameters None.

Return Values: Returns TRUE if successful, otherwise returns FALSE

8.6.1.2.9 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 will optionally block until the setpoint request has been granted.

NOTE

This function is not available on the MX31 platform.

```
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.

8.6.1.2.10 DDKClockSetpointRelease

This function releases a setpoint previously requested using DDKClockSetpointRequest.

NOTE

This function is not available on the MX31 platform.

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

Parameters:

setpoint [in] Specifies the setpoint to be released.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.1.3 DDK_CLK Examples

Example 8-1. Example: CSPDDK Clock Gating

```
    #include "csp.h"    // Includes CSPDDK definitions
```

```
// Enable keypad peripheral clock
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_KPP, DDK_CLOCK_GATE_MODE_ENABLED_ALL);

// Disable keypad peripheral clock
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_KPP, DDK_CLOCK_GATE_MODE_DISABLED);
```

Example 8-2. Example: CSPDDK Clock Rate Query

```
#include "csp.h"    // Includes CSPDDK definitions

UINT32 freq;

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

8.6.2 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.

8.6.2.1 DDK_GPIO Enumerations

Programming Element	Description
DDK_GPIO_PORT	Specifies the GPIO module instance.
DDK_GPIO_DIR	Specifies the direction the GPIO pins.
DDK_GPIO_INTR	Specifies the detection logic used for generating GPIO interrupts.

8.6.2.2 DDK_GPIO Functions

8.6.2.2.1 DDKGpioSetConfig

This function sets the GPIO configuration (direction and interrupt) for the specified pin.

```
BOOL DDKGpioSetConfig(
        DDK_GPIO_PORT port,
        UINT32 pin,
        DDK_GPIO_DIR dir,
        DDK_GPIO_INTR intr)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

dir [in] Direction for the pin.

intr [in] Interrupt configuration for the pin.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.2 DDKGpioBindIrq

This function binds the specified GPIO line with an IRQ that is registered with the OAL to receive interrupts.

```
BOOL DDKGpioBindIrq(
        DDK_GPIO_PORT port,
        UINT32 pin,
        DWORD irq)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

irq [in] Specifies the hardware IRQ that will be translated into a registered SYSINTR within OEMInterruptHandler when the configured interrupt condition for the GPIO line occurs.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.3 DDKGpioUnbindIrq

This function unbinds the specified GPIO line from an IRQ that is registered with the OAL to receive interrupts.

```
BOOL DDKGpioUnbindIrq (
        DDK_GPIO_PORT port,
        UINT32 pin)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.4 DDKGpioWriteData

This function writes the GPIO port data to the specified pins.

```
BOOL DDKGpioWriteData(
        DDK_GPIO_PORT port,
        UINT32 portMask,
        UINT32 data)
```

Parameters

port [in] GPIO module instance.

portMask [in] Bit mask for data port pins to be written.

data [in] Data to be written.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.5 DDKGpioWriteDataPin

This function writes the GPIO port data to the specified pin.

```
BOOL DDKGpioWriteDataPin(
        DDK_GPIO_PORT port,
        UINT32 pin,
        UINT32 data)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

data [in] Data to be written [0 or 1].

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.6 DDKGpioReadData

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

```
BOOL DDKGpioReadData(
        DDK_GPIO_PORT port,
        UINT32 portMask,
        UINT32 *pData)
```

Parameters

port [in] GPIO module instance.

portMask [in] Bit mask for data port pins to be read.

pData [out] Points to buffer for data read.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.7 DDKGpioReadDataPin

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

```
BOOL DDKGpioReadDataPin (
        DDK_GPIO_PORT port,
        UINT32 pin,
        UINT32 *pData)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

pData [out] Points to buffer for data read. Data will be shifted to the LSB.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.8 DDKGpioReadIntr

This function reads the GPIO port interrupt status for the specified pins.

```
BOOL DDKGpioReadIntr(
        DDK_GPIO_PORT port,
        UINT32 portMask,
        UINT32 *pStatus)
```

Parameters

port [in] GPIO module instance.

portMask [in] Bit mask for interrupt status bits to be read.

pStatus [out] Points to buffer for interrupt status.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.9 DDKGpioReadIntrPin

This function reads the GPIO port interrupt status from the specified pin.

```
BOOL DDKGpioReadIntrPin(
        DDK_GPIO_PORT port,
        UINT32 pin,
        UINT32 *pStatus)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

pStatus [out] Points to buffer for interrupt status. Status will be shifted to the LSB.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.2.10 DDKGpioClearIntrPin

This function clears the GPIO interrupt status for the specified pin.

```
BOOL DDKGpioClearIntrPin(
        DDK_GPIO_PORT port,
        UINT32 pin,
        UINT32 *pStatus)
```

Parameters

port [in] GPIO module instance.

pin [in] GPIO pin [0-31].

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.2.3 DDK_GPIO Examples

Example 8-3. Example: CSPDDK GPIO Configuration

```
#include "csp.h"    // Includes CSPDDK definitions

// Configure GPIO1_3 as a level-sensitive interrupt input
DDKGpioSetConfig(DDK_GPIO_PORT1, 3, DDK_GPIO_DIR_IN, DDK_GPIO_INTR_HIGH_LEV);

// Clear interrupt status for GPIO1_3
DDKGpioClearIntrPin(DDK_GPIO_PORT1, 3);

// Bind the GPIO interrupt request to the keypad IRQ registered with the OAL.
// An assertion of the GPIO1_3 interrupt will cause keypad IST to be signaled just
// as if the keypad IRQ was asserting.
DDKGpioBindIrq(DDK_GPIO_PORT1, 3, IRQ_KPP);
```

8.6.3 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.

8.6.3.1 DDK_IOMUX Enumerations

Programming Element	Description
DDK_IOMUX_PIN	Specifies the functional pin name used to configure the IOMUX. The enum value corresponds to the bit offset within the SW_MUX_CTL registers.
DDK_IOMUX_OUT	Specifies the muxing on the output path for a signal.
DDK_IOMUX_IN	Specifies the muxing on the input path for a signal.
DDK_IOMUX_GPR	Specifies the general purpose register (GPR) bits within the IOMUX used to control various muxing features within the SoC.
DDK_IOMUX_PAD	Specifies the functional pad name used to configure the IOMUX. The enum value corresponds to the bit offset within the SW_PAD_CTL registers.
DDK_IOMUX_PAD_SLEW	Specifies the slew rate for a pad.
DDK_IOMUX_PAD_DRIVE	Specifies the drive strength for a pad.
DDK_IOMUX_PAD_MODE	Specifies the CMOS/open drain mode for a pad.
DDK_IOMUX_PAD_TRIG	Specifies the trigger for a pad.
DDK_IOMUX_PAD_PULL	Specifies the pull-up/pull-down/keeper configuration for a pad.

8.6.3.2 DDK_IOMUX Functions

8.6.3.2.1 DDKIomuxSetPinMux

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

```
BOOL DDKIomuxSetPinMux(
        DDK_IOMUX_PIN pin,
        DDK_IOMUX_OUT outMux,
        DDK_IOMUX_IN inMux)
```

Parameters

pin [in] Functional pin name used to select the IOMUX output/input path that will be configured.

outMux [in] Output path configuration.

inMux [in] Input path configuration.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.3.2.2 DDKIomuxGetPinMux

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

```
BOOL DDKIomuxGetPinMux(
        DDK_IOMUX_PIN pin,
        DDK_IOMUX_OUT *pOutMux,
        DDK_IOMUX_IN *pInMux)
```

Parameters

pin [in] Functional pin name used to select the IOMUX output/input path that will be configured.

pOutMux [out] Output path configuration.

pInMux [out] Input path configuration.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.3.2.3 DDKIomuxSetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
        DDK_IOMUX_PAD pad,
        DDK_IOMUX_PAD_SLEW slew,
        DDK_IOMUX_PAD_DRIVE drive,
        DDK_IOMUX_PAD_MODE mode,
        DDK_IOMUX_PAD_TRIG trig,
        DDK_IOMUX_PAD_PULL pull)
```

Parameters

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

slew [in] Slew rate configuration.

drive [in] Drive strength configuration.

mode [in] CMOS/open-drain output mode configuration.

trig [in] Trigger configuration.

pull [in] Pull-up/pull-down/keeper configuration.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.3.2.4 DDKIomuxGetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
        DDK_IOMUX_PAD pad,
        DDK_IOMUX_PAD_SLEW *pSlew,
        DDK_IOMUX_PAD_DRIVE *pDrive,
        DDK_IOMUX_PAD_MODE *pMode,
        DDK_IOMUX_PAD_TRIG *pTrig,
        DDK_IOMUX_PAD_PULL *pPull)
```

Parameters

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

pSlew [in] Slew rate configuration.

pDrive [in] Drive strength configuration.

pMode [in] CMOS/open-drain output mode configuration.

pTrig [in] Trigger configuration.

pPull [in] Pull-up/pull-down/keeper configuration.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.3.2.5 DDKIomuxSetGprBit

This function writes a value into the specified IOMUX GPR bit. These GPR bits are used to control the muxing of signals within the SoC.

```
BOOL DDKIomuxSetGprBit(
        DDK_IOMUX_GPR bit,
        UINT32 data)
```

Parameters

bit [in] GPR bit to be configured.

data [in] Value for the GPR bit [0 or 1].

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.3.3 DDK_IOMUX Examples

Example 8-4. Example: CSPDDK IOMUX Signal Multiplexing

```
#include "csp.h"    // Includes CSPDDK definitions

// Configure the signal multiplexing for GPIO1_3. Route the internal input and
// output path of the GPIO1_3 pin to the GPIO module
DDKIomuxSetPinMux(DDK_IOMUX_PIN_GPIO1_3, DDK_IOMUX_OUT_GPIO, DDK_IOMUX_IN_GPIO);
```

Example 8-5. Example: CSPDDK IOMUX Pad Configuration

```
#include "csp.h"    // Includes CSPDDK definitions

// Configure the GPIO1_3 pad for the following configuration: slow slew rate,
// normal drive strength, CMOS, Schmitt trigger, 100K pull-up.
DDKIomuxSetPadConfig(DDK_IOMUX_PIN_GPIO1_3, DDK_IOMUX_PAD_SLEW_SLOW,
    DDK_IOMUX_PAD_DRIVE_NORMAL, DDK_IOMUX_PAD_MODE_CMOS, DDK_IOMUX_PAD_TRIG_SCHMITT,
    DDK_IOMUX_PAD_PULL_UP_100K);
```

8.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference

The DDK_SDMA interface allows device drivers to allocate, configure, and control shared SDMA resources.

8.6.4.1 DDK_SDMA Enumerations

Programming Element	Description
DDK_DMA_ACCESS	Specifies width of the data for a peripheral DMA transfer.
DDK_DMA_FLAGS	Specifies mode flags within the DMA buffer descriptor.
DDK_DMA_REQ	Specifies DMA request used to trigger SDMA channel execution.

8.6.4.2 DDK_SDMA Functions

8.6.4.2.1 DDKSdmaOpenChan

This function attempts to find an available virtual SDMA channel that can be used to support a memory-to-memory, peripheral-to-memory, or memory-to-peripheral transfers.

```
UINT8 DDKSdmaOpenChan(
        DDK_DMA_REQ dmaReq,
        UINT8 priority,
        LPTSTR lpName,
        DWORD irq)
```

Parameters

dmaReq [in] Specifies the DMA request that will be bound to a virtual channel.

priority [in] Priority assigned to the opened channel.

lpName [in] Not currently used. Set to NULL.

irq [in] Only used if lpName is set to NULL. Specifies the hardware IRQ that will be translated into a registered SYSINTR within OEMInterruptHandler when a transfer interrupt occurs. Set to IRQ_NONE if no interrupt should be generated by the channel.

Return Values: Returns a non-zero virtual channel index if successful, otherwise returns 0.

8.6.4.2.2 DDKSdmaUpdateSharedChan

This function allows a channel that has multiple DMA requests combined into a shared DMA event to be reconfigured for one of the alternate DMA requests.

```
BOOL DDKSdmaUpdateSharedChan(
        UINT8 chan,
        DDK_DMA_REQ dmaReq)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

dmaReq [in] Specifies the DMA request that will be bound to a virtual channel.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.3 DDKSdmaCloseChan

This function closes a virtual DMA channel previously opened by DDKSdmaOpenChan.

```
BOOL DDKSdmaCloseChan(
        UINT8 chan)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan function.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.4 DDKSdmaAllocChain

This function allocates a chain of buffer descriptors for a virtual DMA channel.

```
BOOL DDKSdmaAllocChain(
        UINT8 chan,
        UINT32 numBufDesc)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.5 DDKSdmaFreeChain

This function frees a chain of buffer descriptors previously allocated with DDKSdmaAllocChain.

```
BOOL DDKSdmaFreeChain(
        UINT8 chan)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.6 DDKSdmaSetBufDesc

This function configures a buffer descriptor for a DMA transfer.

```
BOOL DDKSdmaSetBufDesc(
        UINT8 chan,
        UINT32 index,
        UINT32 modeFlags,
        UINT32 memAddr1PA,
        UINT32 memAddr2PA,
        DDK_DMA_ACCESS dataWidth,
        UINT16 numBytes)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

index [in] Index of buffer descriptor within the chain to be configured.

modeFlags [in] Specifies the buffer descriptor mode word flags that control the “continue”, “wrap”, and “interrupt” settings.

memAddr1PA [in] For memory-to-memory transfers, this parameter specifies the physical memory source address for the transfer. For memory-to-peripheral transfers, this parameter specifies the physical memory source address for the transfer. For peripheral-to-memory transfers, this parameter specifies the physical memory destination address for the transfer.

memAddr2PA [in] Used only for memory-to-memory transfers to specify the physical memory destination address for the transfer. Ignored for memory-to-peripheral and peripheral-to-memory transfers.

dataWidth [in] Used only for memory-to-peripheral and peripheral-to-memory transfers to specify the width of the data for the peripheral transfer. Ignored for memory-to-memory transfers.

numBytes [in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.7 DDKSdmaGetBufDescStatus

This function retrieves the status of the “done” and “error” bits from a single buffer descriptor within of a chain.

```
BOOL DDKSdmaGetBufDescStatus(
        UINT8 chan,
        UINT32 index,
        UINT32 *pStatus)
```

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
index	[in] Index of buffer descriptor within the chain.
pStatus	[in] Points to a buffer that will be filled with the status of the buffer descriptor.
Return Values:	Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.8 DDKSdmaGetChainStatus

This function retrieves the status of the “done” and “error” bits from all of the buffer descriptors of a chain.

```
BOOL DDKSdmaGetChainStatus(
        UINT8 chan,
        UINT32 *pStatus)
```

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
pStatus	[in] Points to an array that will be filled with the status of each buffer descriptor in the chain.
Return Values:	Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.9 DDKSdmaClearBufDescStatus

This function clears the status of the “done” and “error” bits within the specified buffer descriptor.

```
BOOL DDKSdmaClearBufDescStatus(
        UINT8 chan,
        UINT32 index)
```

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
index	[in] Index of buffer descriptor within the chain.
Return Values:	Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.10 DDKSdmaClearChainStatus

This function clears the status of the “done” and “error” bits within all of the buffer descriptors of a chain.

```
BOOL DDKSdmaClearChainStatus(
        UINT8 chan)
```

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.11 DDKSdmaInitChain

This function initializes a buffer descriptor chain and the context for a channel. It should be invoked when before a virtual DMA channel is initially started, and when the DMA channel is stopped and restarted.

```
BOOL DDKSdmaInitChain(
        UINT8 chan,
        UINT32 waterMark)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

waterMark [in] Specifies the watermark level used by the peripheral to generate a DMA request. This parameter tells the DMA how many transfers to complete for each assertion of the DMA request. Ignored for memory-to-memory transfers.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.12 DDKSdmaClearChainStatus

This function clears the status of the “done” and “error” bits within all of the buffer descriptors of a chain.

```
BOOL DDKSdmaClearChainStatus(
        UINT8 chan)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.13 DDKSdmaStartChan

This function starts the specified channel.

```
BOOL DDKSdmaStartChan(
        UINT8 chan)
```

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan.

Return Values: Returns TRUE if successful, otherwise returns FALSE.

8.6.4.2.14 DDKSdmaStopChan

This function stops the specified channel.

```
BOOL DDKSdmaStopChan(
        UINT8 chan,
        BOOL bKill)
```

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
bKill	[in] Set TRUE to terminate the channel if it is actively running. Set FALSE to allow the channel to continue running until it yields.
Return Values:	Returns TRUE if successful, otherwise returns FALSE.

Chapter 9
CS8900A Product Ethernet Driver

The Product Ethernet driver is used for connectivity with an IEEE 802.3 Ethernet using the Crystal LAN's CS8900A Ethernet Controller. The driver provides support to communicate with the Ethernet at 10 Mbps speed, as CS8900A is a 10BaseT Ethernet controller. The driver makes use of the persistent information stored in the EEPROM connected to the CS8900A Ethernet controller.

The CS8900A Product Ethernet driver is NDIS 4.0 compliant miniport driver.

9.1 CS8900A Product Ethernet Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31,MX32
MXARM11 CSP Driver Path	N/A
<TGTPLAT> CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\CS8900
Import Library	ndis.lib
Driver DLL	CS8900A.dll
Catalog Item	Catalog –> Third Party –> BSPs –> Freescale <*TGTPLAT*>: ARMV4I –> Device Drivers –> Networking –> Local Area Networking (LAN) devices –> CS8900 - NDIS
SYSGEN Dependency	SYSGEN_NDIS=1 SYSGEN_TCPIP=1 SYSGEN_WINSOCK=1
BSP Environment Variables	BSP_ETHER_CS8900=1

9.2 Requirements

The CS8900A Product Ethernet driver should meet the following requirements:

1. The driver shall conform to the Microsoft Network Driver Interface Specification (NDIS) architecture in Windows CE.
2. The driver shall support IEEE 802.3 Ethernet protocols for communication.

3. The driver shall support two power management modes, full on and full off.

9.3 Hardware Operation

The CS8900A chip is an on-board peripheral which is connected to the processor through the PBC (Peripheral Bus Controller). Refer to the Peripheral Bus Controller CPLD document and CS8900A data sheet for detailed operation and programming information.

The driver reads and makes use of the 6-byte Ethernet MAC address stored at the word addresses 0x000D, 0x000E and 0x000F of the EEPROM. This MAC address can be set and modified using the EBOOT bootloader.

9.3.1 Conflicts with other SoC peripherals

9.3.1.1 i.MX31, MX32 Peripheral Conflicts

No conflicts. (Refer to Peripheral Bus Controller CPLD document for details).

9.4 Software Operation

The Product Ethernet driver follows the Microsoft-recommended architecture for NDIS miniport drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver –> Windows CE Drivers –> Network Drivers –> Network Driver Development Concepts –> Miniports, Intermediate Drivers, and Protocol Drivers**.

9.4.1 Power Management

The power management is currently not implemented for the CS8900A Product Ethernet driver.

9.4.2 Product Ethernet Registry Settings

The following registry keys are required to properly load the CS8900A Product Ethernet driver and to configure the TCP/IP for Ethernet interface. In the following specimen, static IP address is used. To enable dynamic IP address assignment using DHCP, the variable `EnableDHCP` should be set to 1.

```
[HKEY_LOCAL_MACHINE\Comm\CS8900A]
   "DisplayName"="CS8900 Ethernet Driver"
   "Group"="NDIS"
   "ImagePath"="cs8900a.dll"

[HKEY_LOCAL_MACHINE\Comm\CS8900A\Linkage]
   "Route"=multi_sz:"CS8900A1"

[HKEY_LOCAL_MACHINE\Comm\CS8900A1]
   "DisplayName"="CS8900 Ethernet Driver"
   "Group"="NDIS"
   "ImagePath"="cs8900a.dll"

[HKEY_LOCAL_MACHINE\Comm\CS8900A1\Parms]
   ; Change the entries depend on your hardware.
```

```
   ; Do NOT delete BusNumber or BusType, otherwise cs8900a.dll won't be loaded.
   "BusNumber"=dword:0
   "BusType"=dword:0
   ; DuplexMode: 0:AutoDetect; 1:HalfDuplex; 2:FullDuplex.
   "DuplexMode"=dword:0
   ; The Ethernet Physical Address. For example,
   ; Ethernet Address 00:24:20:10:bf:03 is MACAddress1=0024,
   ; MACAddress2=2010,and MACAddress3=bf03.
   ; MACAddress=0000:0000:0000 means to read it from EEPROM.
   ; The MACAddress will be overwritten by MACAddress in EEPROM, if there is a EEPROM.
   "MACAddress1"=dword:1213
   "MACAddress2"=dword:1728
   "MACAddress3"=dword:3121

[HKEY_LOCAL_MACHINE\Comm\CS8900A1\Parms\TcpIp]
   ; This should be MULTI_SZ
   "DefaultGateway"="10.185.60.1"
   ; This should be SZ... If null it means use LAN, else WAN and Interface.
   "LLInterface"=""
   ; Use zero for broadcast address? (or 255.255.255.255)
   "UseZeroBroadcast"=dword:0
   ; Thus should be MULTI_SZ, the IP address list
   "IpAddress"="10.185.60.52"
   ; This should be MULTI_SZ, the subnet masks for the above IP addresses
   "Subnetmask"="255.255.255.0"
   "EnableDHCP"=dword:0

[HKEY_LOCAL_MACHINE\Comm\TcpIp\Parms]

   ;Set to True to keep the device from entering idle mode if there's network adapter
    ;;"NoIdleIfAdapter"=dword:1
    ;Set to True to keep the device from entering idle mode while communicating/loop back
    "NoIdleIfConnected"=dword:1


[HKEY_LOCAL_MACHINE\Comm\Tcpip\Linkage]
   ; This should be MULTI_SZ
   ; This is the list of llip drivers to load
   "Bind"=multi_sz:"CS8900A1"
```

9.5 Unit Test

The CS8900A Product Ethernet driver is tested using the following:

1. Network utilities/operations: Ping to and from the <*TGTSOC*> device, FTP transfers (file put and get) with <*TGTSOC*> device as FTP server and Internet browsing with Pocket Internet Explorer on the <*TGTSOC*> device.
2. Winsock CETK test cases: Winsock 2.0 Test (v4/v6), and Winsock Performance Test with <*TGTSOC*> device as client.

9.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
<*TGTSOC*> board with CS8900A	Board that hosts the CS8900A Product Ethernet driver
PC/machine	To act as counterpart for network operation
An Ethernet or a cross Ethernet cable	To form an Ethernet

9.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
Ws2bvt.dll	Test .dll file for Winsock 2.0 Test (v4/v6).
Perflog.dll	Module that contains functions that monitor and log performance for Winsock Performance Test.
Perf_winsock2.dll	Test .dll file for Winsock Performance Test.
Perf_winsockd2.exe	Test .exe file (server program) for Winsock Performance Test.
Ndt.dll	Protocol driver for One-card network card miniport driver test
Ndt_1c.dll	Test .dll for One-card network card miniport driver test
Ndp.dll	MS_NDP protocol driver for NDIS performance test
Perf_ndis.dll	Test .dll file NDIS performance test

9.5.3 Building the CS8900A Product Ethernet Tests

9.5.3.1 Network utilities related tests

The following registry entries needs to be enabled to allow get/put of files using the anonymous FTP upload, and to be able to access all the files and folders under the root directory on the target:

```
[HKEY_LOCAL_MACHINE\COMM\FTPD]
        "AllowAnonymousUpload" = dword:1
        "DefaultDir" = "\\"
```

For minimum network support and ping to work, the following components need to be enabled in the OS design:

Under `Mobile Handheld[Windows CE devices] –> Communication Services and Networking –>`

```
Networking Features:
Network Driver Architecture (NDIS)
TCP/IP
TCP/IP –> IP Helper API
Winsock Support
```

```
Network Utilities (IpConfig, Ping, Route)
```

For FTP to work, the following components need to be enabled in the OS design:
Under `Mobile Handheld[Windows CE devices] -> Communication Services and Networking -> Servers:`
`FTP Server`

For Video streaming to work, the following components need to be enabled in the OS design:
Under `Mobile Handheld[Windows CE devices] -> Graphics and Multimedia Technologies -> Media:`

```
Media Formats -> AVI Filter
Streaming Media Playback
Video Codecs and Renderers -> Video/Image Compression Manager
Video Codecs and Renderers -> Overlay Mixer
Video Codecs and Renderers -> WMV/MPEG-4 Video Codec
Windows Media Player -> Windows Media Player
Windows Media Player -> Windows Media Player OCX
Windows Media Player -> Windows Media Technologies
Windows Media Player -> Windows Media Technologies -> Windows Media Multicast and Multi-Bit
Rate
Windows Media Player -> Windows Media Technologies -> Windows Media Streaming from Local
Storage
Windows Media Player -> Windows Media Technologies -> Windows Media Streaming over HTTP
```

It will be helpful to add the command line shell and console support:

```
Shell and User Interface -> Shell -> Command Shell:
Command Processor
Command Window
```

9.5.3.2 Winsock 2.0 Test (v4/v6)

The Winsock 2.0 Test (v4/v6) comes pre-built as part of the CETK. No steps are required to build these tests. The Ws2bvt.dll can be found alongside the other required CETK files in the following location:

```
[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I
```

9.5.3.3 Winsock Performance Test

The Winsock Performance Test comes pre-built as part of the CETK. No steps are required to build these tests. The Perf_winsock2.dll and Perf_winsockd2.exe can be found alongside the other required CETK files in the following location:

Perf_winsock2.dll in:

```
[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I
```

Perf_winsockd2.exe in:

```
[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\desktop
```

9.5.3.4 One-Card Network Card Miniport Driver Test

The One-card network card miniport driver test comes pre-built as part of the CETK. No steps are required to build these tests. The ndt.dll and ndt_1c.dll can be found alongside the other required CETK files in the following location:

```
[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I
```

9.5.3.5 NDIS Performance Test

The NDIS performance test comes pre-built as part of the CETK. No steps are required to build these tests. The ndp.dll and perf_ndis.dll can be found alongside the other required CETK files in the following location:

```
[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I
```

9.5.4 Running the CS8900A Product Ethernet Tests

9.5.4.1 Network utilities related tests

9.5.4.1.1 Ping tests

The ping tests can be run as usual from <*TGTSOC*> device as well as from PC side.

9.5.4.1.2 Browsing

The network browsing tests can be done after setting following on the device front panel:

- DNS servers in the TCP/IP properties of CS8900A1 network interface (Control Panel –> Network and Dial-up Connections)
- Proxy server, if used in the network used for test, on the Pocket Internet explorer.

9.5.4.1.3 FTP tests

For running FTP tests, the FTP service needs to be started on the <*TGTSOC*> device using the following command on the DOS prompt:

services start FTP0:

9.5.4.2 Video streaming tests

This can be done by accessing the web sites which provide video clips. An example is: http://www.smartvideo.com. The set-up for Internet Browsing (as mentioned above) is mandatory.

9.5.4.3 Winsock 2.0 Test (v4/v6)

The test can be executed on the <*TGTSOC*> device by using ***tux –o –d Ws2bvt.dl*** in the command line on the <*TGTSOC*>.

For detailed information on the Winsock 2.0 Test (v4/v6) tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Winsock 2.0 Test (v4/v6) in the Platform Builder Help.

9.5.4.4 Winsock Performance Test

We can start the server in on the PC by typing ***Perf_winsockd2 - install*** at the command line. Then client side test executes on the second device by using ***tux –o –d Perf_winsock2.dll –c "-s 10.185.60.56"*** in the

command line on the <*TGTSOC*>, where 10.185.60.56 denotes PC's IP address and needs to be replaced appropriately.

For detailed information on the Winsock Performance tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Winsock Performance Test in the Platform Builder Help.

9.5.4.5 One-Card Network Card Miniport Driver Test

This test can be done by including ndt.dll and ndt_1c.dll in the image, and starting the test by entering ***tux –o –d ndt_1c.dll –c "-t CS8900A1"*** on the command line on the <*TGTSOC*>.

For detailed information on the Winsock Performance tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> One-card Network Card Miniport Driver Test in the Platform Builder Help.

9.5.4.6 NDIS performance test

This test can be done by including ndp.dll and perf_ndis.dll in the image, and starting the test by entering ***tux –o –d perf_ndis.dll –c "CS8900A1"*** on the command line on the <*TGTSOC*>.

For detailed information on the Winsock Performance tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> NDIS Performance Test in the Platform Builder Help

9.6 CS8900A Product Ethernet Driver API Reference

The CS8900A Product Ethernet driver conforms to NDIS 4.0 specification by Microsoft for the miniport network drivers.

Chapter 10
Configurable Serial Peripheral Interface (CSPI) Driver

The Configurable Serial Peripheral Interface (CSPI) module allows rapid data communication with fewer software interrupts than conventional serial communications. The CSPI is equipped with data FIFOs and is a master/slave configurable serial peripheral interface module, capable of interfacing to both SPI master and slave devices.

10.1 CSPI Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information.

10.1.1 For MX31, MX32

Driver Attribute	**Definition**
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\CSPIBUS
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\CSPIBUS
CSP Static Library	<*TGTSOC*>_cspi.lib, mxarm11_cspi.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\CSPIBUS
Import Library	N/A
Driver DLL	cspi.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> CSPI Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_CSPIBUS1=1 BSP_CSPIBUS2=1 BSP_CSPIBUS3=1

10.2 Requirements

The CSPI driver should meet the following requirements:

1. The driver shall support multiple CSPI controllers.
2. The driver shall support the master mode of operation.
3. The driver shall not support the slave mode of operation.

4. The driver shall be a stream interface driver implementing the programming interface defined in this document.
5. The driver shall support two power management modes, full on and full off.

10.3 Hardware Operation

Refer to the chapter on Configurable Serial Peripheral Interface (CSPI) in the hardware specification document for detailed operation and programming information.

10.3.1 Conflicts with other SoC peripherals

10.3.1.1 i.MX31, i.MX32 Peripheral Conflicts

The i.MX31, i.MX32 platforms contains three CSPI modules. The CSPI 1 controller is not connected to any peripheral on the i.MX31/32 ADS and has a pin conflict with the ATA storage device controller. The ATA controller will be enabled in the default BSP configuration.

The CSPI 2 module is connected to the MC13783 module and is controlled by the PMIC driver. The CSPI 2 should not be used with the CSPI bus driver on the i.MX31/32 ADS. The CSPI 2 signals are selected in the IOMUX, as they are required for proper communication with the MC13783 PMIC module.

The CSPI3 has a conflict with UART3 with the following pins

- CSPI3_MOSI
- CSPI3_MISO
- CSPI3_SCLK
- CSPI3_SPI_RDY

CSPI3 has to be configured in Functional mode and UART3 has to be configured in Alternate Mode1.

10.4 Software Operation

10.4.1 DMA Support (provided with iMX31 and i.MX32)

The CSPI driver uses the SDMA controller to exchange data between the different CSPI devices. This minimizes the processing that is required by the ARM core and can also reduce the power consumption during CSPI transfer operations.

Note, however, that the CSPI driver does not always requires the use of DMA support. It means it can work with or without the support of DMA.The CSPI driver has support for an alternative non-DMA. Therefore, the BSP_SDMA_SUPPORT_CSPIx (x = 1,2,3) macro in the bsp_cfg.h header file must be defined to TRUE for enabling DMA based operation and FALSE for non-DMA mode for respective CSPI Channel numbers.

The CSPI driver configuration settings should not be changed without a detailed understanding of the platform’s hardware configuration and operating characteristics. Selecting invalid or incorrect

configuration settings may result in an CSPI driver that will not work properly. Conversely, the CSPI driver performance and resource usage can be fine tuned by adjusting these configuration settings.

The CSPI supports 8-bit, 16-bit, 32-bit access width for all SDMA Read-Write Operations which are achieved with the help of CspiBufRd8, CspiBufRd16, CspiBufRd32, CspiBufWrt8, CspiBufWrt16 and CspiBufWrt32 respectively.

The available compile-time DMA-related configuration options are described in Table 10-1

Configuration Setting Name	Description
CSPI_DMA_WATERMARK_RX, CSPI_DMA_WATERMARK_TX	The transmitter and receiver watermarks that are to be used with CSPI FIFO. The default is 16 for RX and 32 for TX.
CSPI_MAX_DESC_COUNT_TX, CSPI_MAX_DESC_COUNT_RX	Defines the number of DMA buffer descriptors. Currently,CSPI driver does not use scatter-gather list but a statically allocated DMA buffer, so only one buffer descriptor is used for transmit and receive respectively. Default is 1.
CSPI_SDMA_BUFFER_SIZE	This is the size of the DMA buffer used for either transmit or receive operations.

Table 10-1. CSPI Driver Configuration Options

This section will describe the CSPI driver DMA implementation issues and tradeoffs.

In order to use DMA transfers, all DMA data buffers and all DMA buffer descriptors must be properly allocated, managed, and deallocated by the device driver.

The DMA data buffers is allocated from "external memory" (which is provided by off-chip external DRAM). There is flexibility in selecting the size of DMA buffer because typically the size of external memory is large without the need to worry about the possible impact and memory requirements of any other device driver. Memory allocation is handled using standard WinCE system calls.The external memory cannot be placed in low power mode while DMA is active.

The DMA buffer descriptors can also be allocated from either internal or external memory. However, in this case, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). Please refer to the CSPDDK documentation for additional information about the DDKSdmaAllocChain() API.

10.4.2 Communicating Using 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. The basic steps are detailed below.

10.4.3 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 a CSPI port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the CSPI.

1. Append a colon to 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 a 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.

The following code example shows how to open a CSPI port.

```
// Open the SPI port.
hCspi = CreateFile(TEXT("SPI2:"),                 // name of device
          GENERIC_READ|GENERIC_WRITE,             // desired access
          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 a CSPI port, configure the port.

When an application opens a CSPI port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

10.4.4 Configuring the CSPI

Configuring the CSPI port for communications involves the following operations:

- Setting the CSPI frequency for bit count.
- Setting the IOMUX for the corresponding port.
- Creating a CSPI_BUSCONFIG_T object to hold the CSPI bus configurations
- Creating a CSPI_XCH_PKT_T object and fill the packet accordingly.

Before these actions can be taken, a handle to the CSPI port must already be opened. And all the above mentioned four operations have to happen in the Initialization of the CSPI port. The following code example shows how to configure a CSPI port CSPI_BUSCONFIG_T object and CSPI_XCH_PKT_T object.

```
UINT32 m_TxData;       // Buffer to hold Txdata
UINT32 m_RxData;       // Buffer to hold Rxdata
CSPI_BUSCONFIG_T m_BusCnfg;
CSPI_XCH_PKT_T m_XchPkt;
PCSPI_XCH_PKT_T m_pXchPkt = &m_XchPkt;
m_XchPkt.xchEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
// Initialize CSPI bus configuration
m_BusCnfg.chipselect = CSPI_CONREG_SS0;
m_BusCnfg.freq = dwFrequency; // Frequency to set the CSPI bit rate
```

```
m_BusCnfg.bitcount = CSPI_CONREG_BITCOUNT_32BIT; // 32 bit data at a time
m_BusCnfg.sspol = CSPI_CONREG_SSPOL_ACTIVE_HIGH;
m_BusCnfg.ssctl = CSPI_CONREG_SSCTL_ASSERT;
m_BusCnfg.pol = CSPI_CONREG_POL_ACTIVE_HIGH;
m_BusCnfg.pha = CSPI_CONREG_PHA0;
m_BusCnfg.drctl = CSPI_CONREG_DRCTL_DONTCARE;

// Fill entries in exchange packet with static information
m_XchPkt. m_BusCnfg = & m_BusCnfg;
m_XchPkt.pTxBuf = & m_TxData;
m_XchPkt.pRxBuf = & m_RxData;
```

10.4.5 Write Operations

The CSPI driver provides an interface to perform a write operation through the CSPI. The Transmit command writes a packet of data to a target device.

Before this action can be taken, a handle to the CSPI port must already be opened. The write operation requires a call to the **DeviceIoControl** function. As parameters, the CSPI port handle, appropriate IOCTL code, and other input and output parameters are required. Writing is done by issuing a packet with write command. To specify this we need to have a packet structure as follows.

```
typedef struct
{
       unsigned int  data:24;
       unsigned int  null:1;
       unsigned int  address:6;
       unsigned int  rw:1;
} CSPI_PACKET32_FIELDS;

typedef union
{
       CSPI_PACKET32_FIELDS  reg;
       unsigned int          data;
} CSPI_PACKET32;
```

To write to a CSPI port:

1. Create a CSPI_PACKET32 object *cspiWritePacket* and initialize the fields of the packet as follows:
   a) Set the *cspiWritePacket.reg.rw* field as CSPI_WRITE(value 1)
   b) Fill the *cspiWritePacket.reg.data* and *cspiWritePacket.reg.address* field accordingly.
   c) Set *m_TxData* as cspiWritePacket.data;
2. Set the *dwIoControlCode* to the following IOCTL code:

   CSPI_IOCTL_EXCHANGE
3. After calling the **DeviceIoControl** function, call **WaitForSingleObject** to wait on the *xchEvent* object from the *m_XchPkt*. This event will be signalled when the write operation has completed.

The following code example shows how to write to the SPI port.

```
CSPI_PACKET32 packet;
// Format the packet
packet.reg.data = data;              // data to be written to the port
packet.reg.null = 0;
```

```
packet.reg.address = addr;          // address to which data has to be written
packet.reg.rw = CSPI_WRITE;         // Write Command (value 1)

// Write the packet to the TXFIFO
m_TxData = (UINT32) packet.data;    // Send this data to the Buffer

// issue the IOCTL to request a CSPI transfer
DeviceIoControl(hCspi,              // file handle to the driver
                CSPI_IOCTL_EXCHANGE, // I/O control code
                &m_pXchPkt,         // in buffer
                sizeof(m_pXchPkt),  // in buffer size
                NULL,               // out buffer
                0,                  // out buffer size
                0,                  // number of bytes returned
                NULL);              // ignored (=NULL)

// Wait for 1000mSec
WaitForSingleObject(m_XchPkt.xchEvent, CSPI_TIMEOUT_MSEC);Read Operations
```

The CSPI driver provides an interface to perform Read operations over the CSPI bus. The Read from the CSPI needs two stages of operation. First, a write operation is required with *cspiWritePacket.reg.rw* set to CSPI_READ in the transmitter buffer. The target device address must be set in *cspiWritePacket.reg.address* from where the data needs to be read. The next immediate read operation reads the data from the device.

Before these actions can be taken, a handle to the CSPI port must already be opened. It would require a call to the **DeviceIoControl** function. As parameters, the CSPI port handle, appropriate IOCTL code, and other input and output parameters are required. The steps below detail the process of performing a read through the CSPI.

To read from a CSPI port:

1. Create a CSPI_PACKET32 object *cspiWritePacket* and initialize the fields of the packet as follows:
   a) Set the *cspiWritePacket.reg.rw* field as CSPI_READ(value 0)
   b) Fill the *cspiWritePacket.reg.address* field accordingly.
   c) Set *m_TxData* as *cspiWritePacket.data*.
2. Set the *dwIoControlCode* to the following IOCTL code:

   CSPI_IOCTL_EXCHANGE
3. After calling the DeviceIoControl function, call **WaitForSingleObject** to wait on the *xchEvent* object from the *m_XchPkt*. This event will be signalled when the write operation has completed.

The following code example shows how to read from the CSPI port.

```
CSPI_PACKET32 packet;

// Format the packet
packet.reg.data = 0;
packet.reg.null = 0;
packet.reg.address = addr;
packet.reg.rw = CSPI_READ;          // value 0

// Write the packet to the TXFIFO
m_TxData = (UINT32) packet.data;    // Send this data to the Buffer
```

```
// issue the IOCTL to request a CSPI transfer
DeviceIoControl(hCspi,               // file handle to the driver
                CSPI_IOCTL_EXCHANGE, // I/O control code
                &m_pXchPkt,          // in buffer
                sizeof(m_pXchPkt),   // in buffer size
                NULL,                // out buffer
                0,                   // out buffer size
                0,                   // number of bytes returned
                NULL);               // ignored (=NULL)

// Wait for 1000mSec
WaitForSingleObject(m_XchPkt.xchEvent, CSPI_TIMEOUT_MSEC);

// Mask unnecessary fields, and read the data from this buffer
m_RxData &= 0xFFFFFF;
```

10.4.6 Closing the Handle to the CSPI

Call the **CloseHandle** function to close a handle to the CSPI when an application is done using it.

CloseHandle has one parameter which is the handle returned by the **CreateFile** function call that opened the CSPI port.

10.4.7 Power Management

The primary method for limiting power consumption in the CSPI module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. CSPI module needs to be enabled only per transaction. As a result, the CSPI module can be disabled, and its clocks turned off, whenever the module is not processing CSPI packets.

As described in the Write or read operations section, CSPI data transfer operations are handled in `CSPI_XCH_PKT` objects, which contain one or more packets of CSPI data. The CSPI driver turns on the CSPI clocks and enables the CSPI module before processing a `CSPI_XCH_PKT`, and then disables and turns off clocks to the CSPI module after the block of packets has been processed. This limits the time during which the CSPI module is consuming power to the time during which the CSPI is actively performing data transfers.

10.4.7.1 PowerUp

This function is not implemented for the CSPI driver.

10.4.7.2 PowerDown

This function is not implemented for the CSPI driver.

10.4.7.3 IOCTL_POWER_CAPABILITIES

We advertise the power management capabilities with power manager through this IOCTL. Since we support only fully power ON and fully power off, we set DX_MASK(D0) | DX_MASK(D4) as the only power states.

10.4.7.4 IOCTL_POWER_SET

This IOCTL requests a change from one device power state to another. D0 and D4 are the only two supported **CEDEVICE_POWER_STATE** in CSPI driver. So any power change request to the driver other than D0 is D4. Also if the power state is D4 we switch the driver to work in polling mode. And on powering back with D0, we switch to interrupt mode also.

10.4.7.5 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 will not generally issue an **IOCTL_POWER_GET** call to the device unless an application calls **GetDevicePower** with the POWER_FORCE flag set.

10.4.8 CSPI Registry Settings

The following registry key is required to properly load the CSPI device driver for the CSPI1 controller.

```
IF BSP_CSPIBUS1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI1]
       "Prefix"="SPI"
       "Dll"="cspi.dll"
       "Index"=dword:1
ENDIF

IF BSP_CSPIBUS2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI2]
       "Prefix"="SPI"
       "Dll"="cspi.dll"
       "Index"=dword:2
ENDIF

IF BSP_CSPIBUS3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI3]
       "Prefix"="SPI"
       "Dll"="cspi.dll"
       "Index"=dword:3
ENDIF
```

10.5 Unit Test

The serial interface for the CSPI modules are not connected to peripheral devices on the <TGTPLAT> ADS. The unit test for the CSPI driver on the <TGTPLAT> ADS is limited to a manual verification of data and clock signaling measured on a logic analyzer.

10.6 CSPI Driver API Reference

10.6.1 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 been described.

10.6.1.1 CSPI_IOCTL_EXCHANGE

This **DeviceIoControl** request transmits a packet or receives a packet of data to or from the CSPI device. All of the required information should be stored in the CSPI_XCH_PKT_T passed in the *lpInBuffer* field.

Parameters

pInBuffer

Pointer to the CSPI_XCH_PKT_T. This will contain the packet to be read or written. It also holds the number of packets to write and the bus configuration to use for data exchange.

nInBufferSize

Size in bytes of the buffer pointed to by lpInBuffer.

10.6.1.2 CSPI_IOCTL_ENABLE_LOOPBACK

This **DeviceIoControl** request enables the internal loop back of TXFIFO data to the RXFIFO. No parameters are needed.

Parameters

None

10.6.1.3 CSPI_IOCTL_DISABLE_LOOPBACK

This **DeviceIoControl** request disables the internal loop back of TXFIFO data to the RX FIFO. No parameters are needed.

Parameters

None

10.6.2 CSPI Driver Structures

10.6.2.1 CSPI_BUSCONFIG_T

This structure contains the information needed to configure the CSPI bus

```
typedef struct
{
      UINT8     chipselect;
      UINT32    freq;
      UINT8     bitcount;
```

```
        BOOL      sspol;
        BOOL      ssctl;
        BOOL      pol;
        BOOL      pha;
        UINT8     drctl;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Members

chipselect

Selects one of four external SPI Master/Slave Devices. In master mode, these two bits select the external slave devices by asserting the SSn outputs. Only the selected SSn signal will be active while the remaining 3 signals will be negated.

freq

Frequency to which CSIP bus has to be configured.

bitcount

Number of bits to be transmitted at a time.

sspol

Selects the polarity of the chipselect signal.

ssctl

SS Signal waveform select Selects the polarity of the chipselect signal.

pol

SPI clock polarity control.

pha

SPI Clock/Data Control

drctl

SPI Data Ready Control

10.6.2.2 CSPI_XCH_PKT_T

This structure contains the information needed to write or read data using a CSPI port.

```
typedef struct
{
        PCSPI_BUSCONFIG_TpBusCnfg;
        LPVOID    pTxBuf;
        LPVOID    pRxBuf;
        UINT32    xchCnt;
        HANDLE    xchEvent;
} CSPI_XCH_PKT_T, *PCSPI_XCH_PKT_T;
```

Members

pBusCnfg

Object points to the bus configuration of the SPI bus.

pTxBuf

A pointer to a buffer of bytes to transmit in a write operation.

pRxBuf

A pointer to a buffer that will be read into during a read operation.

xchCnt

The number of packets to transmit from pTxBuf / read to pRxBuf.

xchEvent

An event handle that is signalled whenever the packet operation (read or write) has completed. **CreateEvent** must be called to create the event before the packet can be passed to the CSPI.

10.6.2.3 CSPI_PACKET32_FIELDS

This structure contains the format for write or read data using a CSPI port.

```
typedef struct
{
       unsigned int  data:24;
       unsigned int  null:1;
       unsigned int  address:6;
       unsigned int  rw:1;
} CSPI_PACKET32_FIELDS;
```

Members

data

Data that is needed to write to the SPI bus.

null

Unused bit in the register. Should be null.

address

Address from which data to read or write.

rw

Field used to issue read or write command.

10.6.2.4 CSPI_PACKET32

This structure contains the format for write or read data using a CSPI port.

```
typedef union
{
       CSPI_PACKET32_FIELDS    reg;
       unsigned int            data;
} CSPI_PACKET32;
```

Members

reg

Object points to register value for a read or write operation depending on the rw value.

data

Rather than writing the object of CSPI_PACKET32_FIELDS to the port, we prefer to have a 32 bit value, hence this field

Chapter 11
Display Driver

The Windows CE 5.0 and 6.0 BSP display drivers are based on the Microsoft DirectDraw Graphics Primitive Engine (DDGPE) classes and support the Microsoft DirectDraw interface. These drivers combine the functionality of a standard LCD display with DirectDraw support. The display driver interfaces with the Image Processing Unit (IPU). For dumb displays, the IPU Synchronous Display Controller (SDC) combines graphics and video planes and generates display controls with programmable timing. For smart displays, the IPU Post-Processor combines graphics and video planes, and the IPU Asynchronous Display Controller (ADC) generates display controls and programmable timing.

The display driver supports the following display types:

1. Sharp LQ035Q7DB02 QVGA LCD panel.
2. NEC NL6448BC20 VGA LCD panel.
3. Epson LSFxxxxxT00 QVGA Smart LCD.
4. PAL and NTSC TV through the Focus FS453 TV encoder chip.

11.1 Display Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

11.1.1 MX31, MX32ADS

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\IPU\DISPLAY\COMMON, ..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\IPU\DISPLAY\DLL
Import Library	ddgpe.lib, gpe.lib
Driver DLL	ddraw_ipu.dll
Catalog Items	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> IPU Display –> Epson LSFxxxxxT00 (QVGA), NEC NL6448BC20 (VGA), or Sharp LQ035Q7DB02 (QVGA) Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> TV Output FS453/FS454

SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_PP=1 BSP_DISPLAY_NEC_NL6448BC20 = 1 for NEC VGA Panel BSP_DISPLAY_SHARP_LQ035Q7DB02 = 1 for Sharp LCD Panel BSP_DISPLAY_EPSON_LSFxxxxxT00 = 1 for Epson LCD Smart Panel BSP_TVOUT_FOCUS_FS45X = 1 for TV Output

11.2 Requirements

The Display driver should meet the following requirements:

1. The driver shall derive from the DirectDraw Graphics Primitive Engine (DDGPE) class.
2. The driver shall support the DirectDraw Hardware Abstraction Layer (DDHAL).
3. The driver shall support video overlays containing image data in the FOURCC UYVY pixel format.
4. The driver shall support hardware-accelerated color space conversion in video overlays.
5. The driver shall support hardware-accelerated image resizing in video overlays.
6. The driver shall support the Sharp LQ035Q7DB02 QVGA and NEC NL6448BC20 VGA dumb display panels.
7. The driver shall support the Epson LSFxxxxxT00 smart display panel.
8. The driver shall support NTSC and PAL TV through the Focus FS453 TV encoder chip (for Windows CE 5.0 only).
9. The driver shall support two power management modes, full on and full off.

11.3 Hardware Operation

For MX31 and MX32ADS, refer to the chapter on the Image Processing Unit (IPU) in the hardware specification document for detailed operation and programming information.

11.3.1 i.MX31 and i.MX32

11.3.1.1 Conflicts with other SoC peripherals

The display driver supports four display types: Sharp LQ035Q7DB02, NEC NL6448BC20, Epson LSFxxxxxT00, and TV output (NTSC or PAL) (for Windows CE 5.0 only). The TV output display option requires the connection of a daughter card containing a Focus FS453/FS454 TV encoder chip. As the Focus FS45X chip requires a special clocking configuration that differs from the default clocking configuration, some changes must be made on the i.MX31/i.MX32 ADS board to switch between LCD output on the Sharp or NEC panel and TV output through the Focus chip.

To configure the hardware to use the Sharp LQ035Q7DB02, NEC NL6448BC20, or Epson LSFxxxxxT00 panels, the following hardware settings must be verified:

- On the i.MX31/i.MX32 CPU Board, jumper JP1 must be configured to connect pins 1 and 2.
- On the i.MX31/i.MX32 Base Board, set the user switch SW3-4 to ON.

To configure the hardware to generate output to a TV, confirm the following settings:

- On the i.MX31/i.MX32 CPU Board, jumper JP1 must be configured to connect pins 2 and 3.
- On the i.MX31/i.MX32 Base Board, set the user switch SW3-4 to OFF.

A simultaneous TV Out and smart display configuration is allowed when using the Epson LSFxxxxxT00 smart display daughter card. For this configuration to work, the latter set of CPU and Base Board settings (for TV) must be used.

11.3.1.2 Display Control Buttons

11.3.1.2.1 Screen Rotation Button

The S2 hardware button on the MC13783 daughter card has been configured to provide the user a means for changing the screen orientation while the WinCE image is running. Pressing the S2 button toggles the orientation of the screen between a 0 and 270 degree rotation angle.

11.3.1.2.2 TV Output Mode Button

The S3 hardware button on the MC13783 daughter card has been configured to provide the user a means for changing between LCD and TV Output mode. Pressing the S3 button toggles between these two modes.

11.4 Software Operation

11.4.1 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.

11.4.1.1 Using the GDI

The Graphics Device Interface provides basic controls for the display of text and graphics. Please refer to the following help section for information on using the GDI: **Windows CE Features –> Shell and User Interface –> Graphics, Windowing and Events –> GWES Application Development –> Graphics Device Interface.**

11.4.1.2 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. Information on using the DirectDraw API can be found in the following help section: **Windows CE Features –> Graphics and Multimedia Technologies –> Graphics –> DirectDraw**

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

- Page flipping with one backbuffer.
- Overlay surfaces using RGB or the FOURCC UYVY or YV12 pixel formats.

- Overlaying using a color key for the overlay surface for RGB colors (for dumb display panels only).
- Overlaying using a color key for the non-overlay graphics surface for RGB colors.
- Stretching of overlay surfaces.

The IPU contains Post-Processing hardware, which is used within the display driver to accelerate the following operations:

- Color space conversion of UYVY or YV12 overlay data to RGB. This conversion is required in order to combine the overlay data with RGB graphics plane data.
- Resizing of the overlay surface.
- Rotation of the overlay surface (used when the screen orientation is rotated, in Windows CE 5.0 only).
- Resizing and rotation of the primary graphics surface when TV Output mode is enabled and active (Windows CE 5.0 only). This is required to obtain a 640x480 (VGA) or 720x480(D1/NTSC) or 720x576(D1/PAL) resolution image for output to a TV.
- Combining of an overlay surface with a graphics surface. This combining is required for the display of overlay surfaces to a smart panel via the IPU ADC.

11.4.1.3 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 part of the display driver. A detailed description of standard display driver escape codes can be found at the following location in the Platform Builder Help: **Developing a Device Driver** –> **Windows CE Drivers** –> **Display Drivers** –> **Display Driver Escape Codes**.

11.4.2 Configuring the Display

The display is configured based on the **PanelType** registry key, which is described in the **Display Registry Settings** section below. The **PanelType** registry key indicates the display panel that is being used. There are currently four supported display panels: The Sharp LQ035Q7DB02 QVGA LCD panel, NEC NL6448BC20 VGA LCD panel, Epson LSFxxxxxT00 QVGA smart LCD panel, and Toshiba LTM024A60B QVGA smart LCD panel.

11.4.3 Rotation Support

The DirectDraw display driver may be configured to allow screen rotation, through a parameter in the bsp_cfg.h file. If the BSP_DIRECTDRAW_SUPPORT_ROTATION parameter is set to TRUE, the DirectDraw display driver will support rotation. If it is set to FALSE, it will not.

Note: Due to lack of support for DirectDraw and screen rotation (see the Windows CE 5.0 Help, stating that “GDI screen rotation cannot be used with DirectDraw”), a DirectDraw display driver with rotation support enabled may yield more failures in the GDI CETK test suite.

11.4.4 TV Out Configuration

For MX31 and MX32ADS TV output is supported through the Focus FS453/FS454 TV encoder daughter card for Windows CE 5.0. There are four configurable options for TV Out support:

1. NTSC or PAL - Using the TV_NTSC define in the bsp_cfg.h file, a user can select whether to use NTSC or PAL TV output mode. Set TV_NTSC to TRUE to use NTSC mode, or FALSE to use PAL mode.
2. RGB or YUV TV output - Using the TV_YCRCB_INPUT define in the bsp_cfg.h file, a user can select whether to send YUV or RGB data to the TV encoder.
3. VGA or D1 TV output resolution - Using the TV_D1 define in the bsp_cfg.h file, a user can select whether to send VGA (640x480) resolution data or D1 (720x480, 720x576) resolution data to the TV encoder.

11.4.5 Tearing Prevention

When the BSP is configured to use a smart display panel, the display driver may be configured to prevent tearing effects on the display panel by synchronizing updates to the smart panel with a vertical sync signal from the panel. In the bsp_cfg.h file, set the BSP_ADC_ENABLE_TEARING_PREVENTION define to TRUE to enable tearing prevention, and set it to FALSE to disable tearing prevention.

11.4.6 Display Registry Settings

11.4.6.1 MX31

The following registry keys are optionally included, depending on the display panel catalog item included in the OS design.

If the Sharp QVGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; 16bpp
   "VideoBpp"=dword:10           ; RGB565
   "PanelType"=dword:0           ; Sharp QVGA Panel
   "VideoMemSize"=dword:350000   ; 3.5MB
```

If the NEC VGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; 16bpp
   "VideoBpp"=dword:10           ; RGB565
   "PanelType"=dword:1           ; NEC VGA Panel
   "VideoMemSize"=dword:350000   ; 3.5MB
```

If the Epson LSFxxxxxT00 QVGA smart LCD panel is included in the OS design, the following registry keys are also included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; RGB565
   "VideoBpp"=dword:20           ; RGB666 (32bpp internal)
```

```
   "PanelType"=dword:6           ; Epson LSFxxxxxT00 QVGA Smart Panel
   "VideoMemSize"=dword:350000   ; 3.5MB
```

If TV Output is included in the OS design (only for Windows CE 5.0), the following registry keys are also included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "TVSupported"=dword:1         ; NTSC TV out mode supported
   "VideoMemSize"=dword:400000   ; 4.0MB
```

The **VideoMemSize** is set to a value sufficiently high to permit the allocation of memory for several surfaces: The primary graphics surface, two video overlay surfaces, a surface for the output of combining operations (if a smart display panel is used), a surface used for TV output (if TV output is enabled) and additional surfaces created by applications, such as the Windows CE Media Player.

Bpp refers to the bits per pixel of the User Interface (UI) surface. Currently both the NEC and VGA display panels require 16-bit RGB pixel data, so the **Bpp** for these panels is 0x10, or 16. For the Epson smart display panel, either 16 or 32 bits per pixel may be used.

VideoBpp refers to the bits per pixel of the video overlay surface. It may be preferred to keep a greater pixel depth for video data in order to ensure high quality video. This value may be set to 16 bits per pixel or 32 bits per pixel (RGB888 unpacked).

11.4.6.2 MX32ADS

The following registry keys are optionally included, depending on the display panel catalog item included in the OS design.

If the Sharp QVGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; 16bpp
   "VideoBpp"=dword:10           ; RGB565
   "PanelType"=dword:0           ; Sharp QVGA Panel
```

If the NEC VGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; 16bpp
   "VideoBpp"=dword:10           ; RGB565
   "PanelType"=dword:1           ; NEC VGA Panel
```

If the Epson LSFxxxxxT00 QVGA smart LCD panel is included in the OS design, the following registry keys are also included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                ; RGB565
   "VideoBpp"=dword:20           ; RGB666 (32bpp internal)
   "PanelType"=dword:6           ; Epson LSFxxxxxT00 QVGA Smart Panel
```

If TV Output is included in the OS design (only for Windows CE 5.0), the following registry keys are also included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "TVSupported"=dword:1         ; NTSC TV out mode supported
```

Unlike the MX31 platform, which uses the **VideoMemSize** registry key to specify the video memory size, the MX32ADS video memory size is specified in the image_cfg.h file. This is because separate regions of memory are reserved in the MX32ADS for use by the display driver. These regions are defined using the following constants: IMAGE_WINCE_PP_RAM_SIZE and IMAGE_WINCE_IPU_RAM_SIZE. The IMAGE_WINCE_PP_RAM_SIZE must be large enough to hold two video overlay surfaces. The IMAGE_WINCE_IPU_RAM_SIZE must be large enough to hold the following surfaces: The primary graphics surface, an additional surface for combining output (if the Epson smart panel is enabled), a surface used for TV output (if TV output is enabled) and additional surfaces created by applications, such as the Windows CE Media Player.

Bpp refers to bits per pixel. Currently both the NEC and VGA display panels require 16-bit RGB pixel data, so the **Bpp** for these panels is 0x10, or 16. For the Epson smart display panel, either 16 or 32 bits per pixel may be used.

VideoBpp refers to the bits per pixel of the video overlay surface. It may be preferred to keep a greater pixel depth for video data in order to ensure high quality video. This value may be set to 16 bits per pixel or 32 bits per pixel (RGB888 unpacked).

11.4.7 Power Management

The display driver consumes power primarily through the operation of various IPU sub-modules, such as the SDC, which combines and displays video and graphics data, and through the operation of the display panel. To facilitate management of these modules, the display driver implements the power management I/O Control (IOCTL) code IOCTL_POWER_SET.

Since display refresh operations are manually executed with a smart display panel, additional power savings are achieved when a smart panel is used by disabling the ADC and DI modules in between display updates.

11.4.7.1 PowerUp

This function is not implemented for the display driver.

11.4.7.2 PowerDown

This function is not implemented for the display driver.

11.4.7.3 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 display panel is enabled. If a dumb LCD is in use, the IPU's Display Interface (DI) and SDC modules are enabled.

- D4 – If a dumb LCD is in use, the DI and SDC modules of the IPU are disabled. The display panel is disabled.

11.5 Unit Test

The display driver is subject to two test suites provided with the Windows CE Test Kit (CETK): the Graphics Device Interface (GDI) Test and the DirectDraw Test. Additionally, video playback may be verified by 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.

11.5.1 Unit Test Hardware

The following table lists the required hardware to run the GDI and DirectDraw tests.

Requirements	Description
SHARP LQ035Q7DB02 QVGA Panel or NEC NL6448BC20 VGA Panel or Epson LSFxxxxxT00 QVGA Panel	Display panel required for display of graphics data.

11.5.2 Unit Test Software

11.5.2.1 GDI Tests

The following table lists the required software to run the GDI tests.

Requirements	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.

11.5.2.2 DirectDraw Tests

The following table lists the software required to run the DirectDraw tests:

Requirements	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

11.5.2.3 Windows Media Player Tests

The following table lists the software required to perform WMV playback with Windows Media Player:

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

11.5.3 Building the Display Tests

The GDI and DirectDraw tests come pre-built as part of the CETK. No steps are required to build these tests. The DDrawTK.dll, Gdiapi.dll, and Ddi_test.dll files can be found alongside the other required CETK files in the following locations:

for Windows CE 5.0 - [Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

for Windows CE 6.0 - [Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4i

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.

11.5.4 Running the Display Tests

11.5.4.1 Running the GDI Tests

The command line for running the GDI tests is ***tux –o –d gdiapi.dll***.

For detailed information on the GDI tests and command line options for these tests, see **Debugging and Testing** –> **Tools for Debugging and Testing** –> **Windows CE Test Kit** –> **CETK Tests** –> **DirectDraw Test** –> **Modifying the Graphics Device Interface Test** in the Platform Builder Help.

The following table describes the test cases contained in the GDI test suite:

Test case	Description
100-104: Clip	Tests the functionality of clipping using different shapes and verifies the functionality of complex clip regions.
200-231: Draw	Calls functions that draw and functions that apply complex effects to drawing. These test cases perform blitting, line drawing, filling, color table manipulation, bitmap type creation, and device attribute modification. **NOTE**: Test case 231, AlphaBlend, is skipped, as it is not supported in the image.
300-307: Palette	Verifies color matching and color conversion for palettes, and modifies associated palettes.
500-512: Region	Tests region management by calling functions that modify region rectangles.
600-608: Brush and pen	Assesses the functionality of brushes and brush alignments.
700-710: Device attribute	Verifies device attributes and exercises functions that modify device attributes.
800-808: Device context	Creates, retrieves, saves, and restores a device context.
900-905: Device object	Calls functions that retrieve, modify, and delete GDI objects.
1000-1011: Font	Verifies font enumeration, selection, and attributes.
1100-1108: Text	Writes text to various locations on the display. If the font required by the test is not available, some test cases do not run.
1200-1205: Print	Passes bad parameters to printing functions.
1300-1303: Verify	Assesses the functionality of test verification functions such as **CheckScreenHalves** and **CheckAllWhite**.
1400, 1401: Manual	Manually tests font drawing. You can use these test cases to exercise code paths. To step through these test cases, press the left SHIFT key.

11.5.4.2 Running the DirectDraw Tests

The command line for running the DirectDraw tests is ***tux –o –d ddrawtk***.

For detailed information on the DirectDraw tests and command line options for these tests, see **Debugging and Testing** –> **Tools for Debugging and Testing** –> **Windows CE Test Kit** –> **CETK Tests** –> **DirectDraw Test** –> **Modifying the DirectDraw Test** in the Platform Builder Help.

The following table describes the test cases contained in the DirectDraw test suite:

Test case	Description
100: Get Caps	Retrieves the capabilities of the hardware abstraction layer (HAL), verifies that the operation is successful, and then displays all capabilities retrieved. This test case fails if it cannot retrieve the capabilities of the HAL.
101: Enumerate Display Modes	Enumerates the DirectDraw display modes, verifies that the enumeration completes, and then displays the results of the enumeration.
200: Blt (Windowed Mode)	Executes blits to and from various surfaces. The test case verifies that the actual destination matches the expected destination for each blit. This test case fails if any blits are unsuccessful.

Test case	Description
210: ColorKey Blt (Windowed Mode)	Executes a variety of color key blits to and from assorted surfaces. The test case verifies that the actual destination matches the expected destination for each test. This test case fails if any color key blits are unsuccessful.
220: Color Filling Blts (Windowed Mode)	Executes a variety of color fill blits to assorted surfaces. The test case verifies that the surface is filled with the specified color. This test case fails if any color fill blits are unsuccessful.
300: Blt (Exclusive Mode)	Executes a variety of blits to and from assorted surfaces. The test case verifies that the actual destination matches the expected destination for test. This test case fails if any blits are unsuccessful.
310: ColorKey Blt (Exclusive Mode)	Executes a variety of color key blits to and from assorted surfaces. The test case verifies that the actual destination matches the expected destination for each test. This test case fails if any color key blits are unsuccessful.
320: Color Filling Blts (Exclusive Mode)	Executes a variety of color fill blits to assorted surfaces. The test case verifies that the surface is filled with the specified color. This test case fails if any color fill blits are unsuccessful.
330: Flip (Exclusive Mode)	Executes a variety of blits to a flipping chain and verifies that the flips are successful and that all surfaces display correctly. This test case fails if any flips or surface verifications fail.
400: CreateVideoPort (Video Port Container Test)	Enumerates the available video ports and connections for the video ports. This test case then verifies that each enumerated connection can be created.
410: EnumVideoPorts (Video Port Container Test)	Enumerates the available video ports and then enumerates the video ports based on specific capabilities.
420: GetVideoPortConnectInfo (Video Port Container Test)	Verifies that the **GetVideoPortConnectInfo** function appropriately handles a variety of input conditions.
430: QueryVideoPortStatus (Video Port Container Test)	Verifies that the **QueryVideoPortStatus** function appropriately handles each video port.
500: GetBandwidthInfo (Video Port Test)	Verifies that the **GetBandwidthInfo** function returns consistent information about each type of video port available.
502: GetSetColorControls (Video Port Test)	Verifies that the **GetColorControls** and **SetColorControls** functions set and return consistent color controls if color control is supported.
504: GetInputOutputFormats (Video Port Test)	Verifies that the **GetInputFormats** and **GetOutputFormats** functions return consistent information under a variety of calling conditions.
506: GetFieldPolarity (Video Port Test)	Verifies that the **GetFieldPolarity** function returns consistent information about the field polarity of a video port.
508: GetVideoLine (Video Port Test)	Verifies that the **GetVideoLine** function returns consistent information.
510: GetVideoSignalStatus (Video Port Test)	Verifies that the **GetVideoSignalStatus** function returns consistent information.
512: SetTargetSurface (Video Port Test)	Verifies that the **SetTargetSurface** function appropriately handles improper input.
514: StartVideo (Video Port Test)	Verifies that calls to the **StartVideo** function succeed with a variety of flags set.
516: StopVideo (Video Port Test)	Verifies that a call to the **StopVideo** function succeeds.
518: UpdateVideo (Video Port Test)	Verifies that calls to the **UpdateVideo** function succeed with a variety of flags set.
520: WaitForSync (Video Port Test)	Verifies that the **WaitForSync** function behaves consistently with each possible flag.
1200: Blt (Interactive Windowed Mode)	Executes blits that include color fills and scaling. You must manually verify that each blit is successful.

Test case	Description
1240: Overlay Blt (Interactive Windowed Mode)	Executes blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. If the test succeeds, an overlay appears in the middle of the screen with a blue and yellow checkerboard pattern. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1250: ColorKeyOverlay Blt (Interactive Windowed Mode)	Executes color key blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. If the test succeeds, a blue checkerboard pattern appears in a variety of locations across the display of the target device. If the driver enables stretching, the test also stretches the checkerboard pattern. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1260: ColorFill Overlay Blt (Interactive Windowed Mode)	Executes color fill blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. This test case cycles through red, green and blue on the primary overlay surface. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1300: Blt (Interactive Exclusive Mode)	Executes blits that you must verify when prompted. The test does not contain an automated mechanism for verifying scaling.
1340: Overlay Blt (Interactive Exclusive Mode)	Executes blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. If the test succeeds, an overlay is located in the middle of the screen on the target device with blue and yellow horizontal lines. The primary surface behind the overlay surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1350: ColorKeyOverlay Blt (Interactive Exclusive Mode)	Executes color key blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. If the test succeeds, blue horizontal bars appear across the display of the target device. The primary surface behind the overlay surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1360: ColorFill Overlay Blt (Interactive Exclusive Mode)	Executes color fill blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so you should manually verify that the output is correct. This test case cycles through red, green and blue on the primary surface and the overlay surface. The color fills on the primary surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.

NOTE

The following DirectDraw test cases are not supported by the DirectDraw driver, and are therefore skipped: 410, 420, 430, 500, 502, 504, 506, 508, 510, 512, 514, 516, 518, 520.

11.5.4.3 Running the Windows Media Player tests

The command line for starting playback of a WMV test video clip in Windows Media Player is ***ceplayer [wmv test file]*** (e.g. “ceplayer motocross_208x160_30fps.wmv”). If audio support is not included in the current BSP, a dialog box reading “Audio hardware is missing or disabled” will pop up when the WMV file is being loaded. Select OK to continue to WMV playback.

Correct operation of this test is confirmed by observing the application and verifying that the video clip is playing at a smooth rate (it should not be dropping frames or otherwise appearing jerky) with a clear image, normal coloring, and correct image sizing.

11.6 Display Driver API Reference

Documentation for the display driver APIs can be found within the Platform Builder Help. No additional custom API information is required for the features currently supported in the display driver.

Reference information on basic display driver functions, methods, and structures can be found at the following location in the PB Help documentation:

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

Reference information on DirectDraw functions, callbacks, and structures can be found at the following location in the PB Help documentation:

Windows CE Features –> Graphics and Multimedia Technologies –> Graphics –> DirectDraw

Chapter 12
Dynamic Voltage and Frequency Control (DVFC)

The BSP includes a component called the Dynamic Voltage and Frequency Control (DVFC) driver that provides combined support for DVFS (Dynamic Voltage Frequency Scaling) and DPTC (Dynamic Process Temperature Compensation). 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 can respond to DVFC and DPTC hardware logic that is monitoring CPU loading and process/temperature performance of the silicon.

12.1 DVFC Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	...\CSP\ARM\FREESCALE\<*TGTPLAT*>\DRIVERS\DVFC
CSP Static Library	<*TGTSOC*>_dvfc.lib
Platform Driver Path	...\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\DVFC\
Import Library	dvfc_mc13783.lib
Driver DLL	dvfc_mc13783.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*>–>Device Drivers–> MC13783 DVFC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PMIC_MC13783 = 1 BSP_DVFC_ MC13783 = 1

12.2 Requirements

1. Execute as a device driver and provide synchronized support of the DPTC and DVFS power management features.
2. MX31 will support both DPTC and DVFS. MX32 will only support DVFS.
3. Support integer frequency scaling on MX31. Support DVFS with PLL switch on MX32.
4. Provide integrated voltage control supplied from MC13783.

5. Expose stream interface for initialization and power management.
6. Support D0 and D4 driver power states for all targets. MX32 will support control of frequency/voltage setpoint based on Power Manager device power states.

12.3 Hardware Operation

Refer to the <*TGTSOC*> hardware specification document for detailed operation and programming information.

12.3.1 i.MX31 ADS Configuration

The DVFC driver is dependent upon the MC13783 interface for dynamic voltage control. The MC13783 board must be present on the i.MX31 ADS. The i.MX31 CPU board must be configured to source power from the MC13783. Refer to the **Windows CE BSP for i.MX31 User's Guide** for the proper board configuration.

12.3.2 i.MX32 ADS Configuration

The DVFC driver is dependent upon the MC13783 interface for dynamic voltage control. The MC13783 board must be present on the i.MX32 ADS. The i.MX32 CPU board must be configured to source power from the MC13783. Refer to the **Windows CE BSP for i.MX32 User's Guide** for the proper board configuration.

12.3.3 Conflicts with other SoC peripherals

12.3.3.1 i.MX31 Peripheral Conflicts

The signals used for the dynamic voltage control interface between i.MX31 and MC13783 cannot be used for other purposes. In particular, the i.MX31 GPIO1_5 pin is used by the DVFS hardware to determine when the new voltage setting has been reached. This pin should not be configured for GPIO purposes.

12.3.3.2 i.MX32 Peripheral Conflicts

The signals used for the dynamic voltage control interface between i.MX32 and MC13783 cannot be used for other purposes. In particular, the i.MX32 GPIO1_5 pin is used by the DVFS hardware to determine when the new voltage setting has been reached. This pin should not be configured for GPIO purposes.

12.4 Software Operation

12.4.1 Loading and Initialization

The DVFC driver will be automatically loaded by device manager as a stream driver. As part of the loading procedure of stream drivers, the device manager will invoke 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.

12.4.2 i.MX31 Software Operation

This section will describe the software operation for the i.MX31 DVFC driver.

12.4.2.1 Tuning Hardware Load Tracking for i.MX31

The DVFC driver utilizes the hardware load tracking available within the i.MX31 DVFS logic. The load tracking hardware monitors the CPU activity and notifies the system to adjust the DVFS setting to meet the required CPU performance. By adjusting parameters used to configure the load tracking hardware, users can control the CPU loading characteristics that trigger DVFS transitions. These parameters are configured using DVFC driver registry keys. Refer to the Registry Settings section in this chapter for more information.

12.4.2.2 External Interface for i.MX31

The DVFC driver for i.MX31 allows other drivers/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. Refer to the CSPDDK documentation for the following functions:

1. DDKClockEnablePanicMode
2. DDKClockDisablePanicMode
3. DDKClockBusScale

12.4.2.3 Registry Settings for i.MX31

This section describes the registry keys supported by the DVFC driver.

Key Name	Key Type	Description
`ThreadPriority`	`REG_DWORD`	This key specifies the thread priority assigned to the DVFC interrupt service thread. **Note:** If this key is missing, a default thread priority of 250 will be used.
`TrackingLoadDown`	`REG_DWORD`	This key specifies the CPU loading threshold (expressed as a percentage) used by the load tracking hardware to determine when a lower frequency/voltage setting is requested. If the average CPU load is less than `TrackingLoadDown` over a continuous sample window specified by the `TrackingWindowDown` registry key, the DVFS logic will request a lower frequency/voltage setting. **Note:** If this key is missing, a default CPU load of 50% will be used as the threshold.
`TrackingLoadUp`	`REG_DWORD`	This key specifies the CPU loading threshold (expressed as a percentage) used by the load tracking hardware to determine when a higher frequency/voltage setting is requested. If the average CPU load is greater than `TrackingLoadUp` over a continuous sample window specified by `TrackingWindowUp` registry key, the DVFS logic will request a higher frequency/voltage setting. **Note:** If this key is missing, a default CPU load of 90% will be used as the threshold.

Key Name	Key Type	Description
TrackingWindowDown	REG_DWORD	This key specifies the continuous sample window (expressed in msec) used by the load tracking hardware to determine when a lower frequency/voltage setting is requested. If the average CPU load is less than the TrackingLoadDown registry key over a continuous sample window specified by TrackingWindowDown, the DVFS logic will request a lower frequency/voltage setting. **Note:** If this key is missing, a default tracking window of 5 msec will be used.
TrackingWindowUp	REG_DWORD	This key specifies the continuous sample window (expressed in msec) used by the load tracking hardware to determine when a higher frequency/voltage setting is requested. If the average CPU load is greater than the TrackingLoadUp registry key over a continuous sample window specified by TrackingWindowUp, the DVFS logic will request a higher frequency/voltage setting. **Note:** If this key is missing, a default tracking window of 5 msec will be used.
TrackingEMAC	REG_DWORD	The load tracking hardware computes an exponential moving average (EMA) of the tracked CPU load. This key specifies the number of samples that are included in the EMA calculation. A lower number of EMA samples will decrease the hysteresis in the EMA load tracking calculation. Conversely, a higher number of EMA samples will increase the hysteresis in the EMA load tracking calculation. Refer to the i.MX31 hardware manual for more information and available settings. This parameter will be used to update the EMAC bits of the LTR2 register in the i.MX31 CCM. **Note:** If this key is missing, a default setting of 31 samples (EMA code = 0x10) will be used.
DvfsForceOff	REG_DWORD	Setting this key to a non-zero value will force the DVFS functionality of the DVFC driver to be disabled. **Note:** If this key is missing, DVFS functionality will be enabled by default.
DptcForceOff	REG_DWORD	Setting this key to a non-zero value will force the DPTC functionality of the DVFC driver to be disabled. **Note:** If this key is missing, DPTC functionality will be enabled by default.
DvsSpeed	REG_DWORD	This key specifies the transition rate for the output voltage supplied from the MC13783 PMIC. Refer to the MC13783 hardware manual for more information and available settings. This parameter will be used to update the DVSSPEED selection bits of the MC13783. **Note:** If this key is missing, a default setting of 16 us / 25 mV step (DVSSPEED = 0x3) will be used.
LowSpeedVolt	REG_DWORD	This key specifies the fixed voltage supplied to the CPU from the MC13783 PMIC for the low-speed CPU DVFS setpoint. The voltage is expressed as a coded value that reflects the output voltage selections available from the MC13783 buck switchers. Refer to the MC13783 hardware manual for more information and available settings. This parameter will be used to update the SW1ADVS selection bits of the MC13783. **Note:** If this key is missing, a default of 1.350V (SW1ADVS = 0x12) will be used.

Key Name	Key Type	Description
LowBusVolt	REG_DWORD	This key specifies the fixed voltage supplied to the CPU from the MC13783 PMIC for the low-speed CPU and low-speed bus (activated by DDKClockScaleBus) DVFS setpoint. The voltage is expressed as a coded value that reflects the output voltage selections available from the MC13783 buck switchers. Refer to the MC13783 hardware manual for more information and available settings. This parameter will be used to update the SW1ADVS selection bits of the MC13783. **Note:** If this key is missing, a default of 1.275V (SW1ADVS = 0x0F) will be used.
Verbose	REG_DWORD	Setting this key to a non-zero value will enable retail messages within the DVFC driver. The retail messages will provide information regarding DVFS and DPTC configuration and setpoint transitions. **Note:** If this key is missing, retail messages will be disabled by default.

12.4.2.4 Power Management for i.MX31

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 too must support the Power Manager device driver interface. This allows the DVFC driver to be notified of when the system is suspending/resuming and configure the DVFS and DPTC hardware blocks appropriately.

12.4.2.4.1 PowerUp

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

12.4.2.4.2 PowerDown

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

12.4.2.4.3 IOCTL_POWER_CAPABILITIES

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

12.4.2.4.4 IOCTL_POWER_SET

The DVFC driver supports requests to enter D0-D4 device power state. Power states D1-D3 are treated as D4.

12.4.2.4.5 IOCTL_POWER_GET

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

12.4.3 i.MX32 Software Operation

The i.MX32 DVFC driver is the central point in the BSP for controlling voltage and frequency scaling. DVFC communicates with the PMIC and MX32 CCM to coordinate DVFS. The DVFC driver responds to setpoint requests from DDK_CLK (via driver calling **DDKClockSetGatingMode**) and Power Manager

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

12.4.3.1 Voltage/Frequency Setpoints

The i.MX32 DVFC driver supports 4 voltage frequency setpoints. The following table provide the voltage/frequency characteristics for these setpoints.

Table 1: i.MX32 DVFC Setpoints

Setpoint Name	CPU/BUS/PER Clock [MHz]	Core Voltage
DDK_DVFC_SETPOINT_TURBO	528/132/66	1.425 V
DDK_DVFC_SETPOINT_HIGH	396/132/66	1.200 V
DDK_DVFC_SETPOINT_MEDIUM	132/66/66	1.000 V
DDK_DVFC_SETPOINT_LOW	132/33/33	1.000 V

The setpoint attributes are controlled by the definitions in the DVFC driver code (found in \PLATFORM\MX32ADS\SRC\DRIVERS\DVFC\COMMON\dvfc.c). The DVFC driver uses these definitions to populate a global setpoint array (g_SetPointConfig) that will be referenced during setpoint transitions.

12.4.3.2 Setpoint Mapping

The i.MX32 peripherals may not be able to operate properly in all of the supported setpoints due to minimum frequency/voltage requirements. Therefore, drivers that support these peripherals need a method of communicating setpoint requirements. Drivers communicate clocking and setpoint requirements through the use of APIs in the CSPDDK. The mapping of DDK_CLK clock management routines (DDKClockSetGatingMode) to DVFC setpoints is located in **UpdateSetpointRequestCount** (found in \PLATFORM\MX32ADS \SRC\DRIVERS\CSPDDK\DDK_CLK\ddk_clk.c). To change the setpoint mapping for a specific peripheral, modify the code in **UpdateSetpointRequestCount**.

WARNING

Do not map a peripheral to a setpoint that violates the electrical specification of the i.MX32 or does not provide adequate clocking for the peripheral protocol specification.

The DVFC driver advertises support for IOCTL_POWER requests from Power Manager. A IOCTL_POWER_SET request will be 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 \MX32ADS\SRC\DRIVERS\DVFC\COMMON\dvfc.c). To change the setpoint mapping for a specific device power state (D0-D4), modify the code in **DvfcMapDevPwrStateToSetpoint**.

12.4.3.3 EMI DCG (Dynamic Clock Gating)

The i.MX32 provides hardware support for dynamically gating clocks to the EMI in the absence of accesses to the EMI interface. This feature can be enabled and configured in the BSP using **BSP_DVFC_EMI_DCG** definitions in \PLATFORM\MX32ADS\SRC\INC\bsp_cfg.h. Refer to the i.MX32 hardware documentation for more information about this feature.

12.4.3.4 Frequency Scaling Operation

The i.MX32 DVFC driver supports frequency scaling of the MCU, MAX (AHB), IPG (peripheral), HSP (IPU), and NFC clock domains. This section will describe the main features of the frequency scaling.

12.4.3.4.1 Dual PLL DVFS

The i.MX32 has the ability to switch between the MCU PLL and serial PLL during a DVFS transition. Dual PLL DVFS allows the non-integer multiples of the CPU clock rate between setpoints (528/396) and saves power by allowing the PLL output clock to be matched exactly to the desired CPU clock rate.

NOTE

The serial PLL will be used by the DVFC driver to implement DVFS. If the DVFC driver is activated, the serial PLL cannot be selected as a baud clock source for peripherals. Drivers that specify the baud clock configuration using DDKClockConfigBaud must not select the serial PLL as the input clock source when using DVFC.

12.4.3.4.2 DDR Bus Scaling

Scaling the AHB bus frequency with DDR memory requires a calibration sequence to be executed from an internal uncached memory space. The calibration sequence must disable interrupts to prevent bus masters from accessing the DDR during the sequence. A custom kernel IOCTL (**IOCTL_HAL_DVFC_BUS_SCALE**) called from the DVFC driver is used to perform DDR delay line calibration. The DDR calibration sequence is executed from uncached IRAM defined by **IMAGE_WINCE_DVFC_IRAM** definitions in \PLATFORM\MX32ADS\SRC\INC\image_cfg.h.

12.4.3.4.3 Peripheral Clock Scaling

Transitions to/from the LOW setpoint require the IPG clock to be scaled (IPG clock is scaled between 66 MHz and 33 MHz). The BSP is configured to use IPG clock as the PERCLK source. This implies that scaling the IPG clock will impact peripherals that use PERCLK. Data transmission clocks (communications peripherals) or tick rate (timer peripherals) will be altered if PERCLK is scaled.

The DVFC driver will not scale the IPG clock unless all peripherals that depend on PERCLK are inactive. The activity of peripherals using PERCLK is determined by **DDK_CLK_CONFIG** flags set by the CSPDDK. Drivers need not be concerned about PERCLK being changed while clocks have been activated using **DDKClockSetGatingMode**. However, after peripheral clocks have been deactivated using **DDKClockSetGatingMode**, the driver must assume that PERCLK could have been altered upon the next attempt to activate peripheral clocks. The altered PERCLK frequency can be determined by invoking **DDKClockGetFreq** after activating clocks for the peripheral via **DDKClockSetGatingMode** (calling

DDKClockGetFreq after enabling peripheral clocks ensures the PERCLK rate will not incur additional changes).

Peripherals that use synchronous clocking (i.e. SDHC, CSPI, etc.) can continue to operate at a reduced PERCLK rate without changes to the data transmission clock rate. However, peripherals that use asynchronous clocking (i.e. UART) or have specific clocking requirements (i.e. ATA to meet UDMA timing) cannot tolerate PERCLK rate changes between transfers. In such cases, the CSPDDK source code can be modified to restrict setpoints for specific peripherals. Refer to Section 12.4.3.2, "Setpoint Mapping" for more information regarding the mapping between DVFC setpoints and peripheral clocking. Upon invoking **DDKClockSetGatingMode,** the DVFC will coordinate a setpoint transition, if necessary, to grant the driver a voltage/frequency setpoint that meets or exceeds the request.

Timer peripherals (EPIT, GPT, etc.) that are configured to clock from PERCLK would experience clock skew/drift during IPG clock scaling. However, this can be eliminated by using CKIL as the input clock source. This could result in a significant reduction in the available clock resolution (PERCLK runs much faster than CKIL) but decouples the hardware timer from IPG clock scaling events.

The OAL support for the OS tick timer using EPIT1 has been extended to support the CKIL clock input and eliminate skew/drift resulting from scaling PERCLK. A dedicated OAL library to support this functionality can be integrated into the OS image using the **BSP_OAL_TIMER32K** environment variable. Refer to the **Windows CE BSP for MCIMX32 (i.MX32) ADS User's Guide** for more information regarding the **BSP_OAL_TIMER32K** configuration.

12.4.3.5 Configuring BSP for Low Setpoint

Here are the steps required to allow the BSP to enter the lowest DVFC setpoint:

1. Add MX32ADS BSP "MC13783 DVFC" to the MX32ADSMobility OSDesign.
2. Add BSP_OAL_TIMER32K to the MX32ADSMobility Platform...Settings...Environment.
3. To get the display and backlight to turn off more quickly, add the following registry information to the bottom of platform.reg:

```
[-HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts]

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts]
        "ACUserIdle"=dword:0A      ; 10 seconds
        "ACSystemIdle"=dword:0A    ; 10 seconds
        "ACSuspend"=dword:0        ; never suspend
        "BattUserIdle"=dword:0A    ; 10 seconds
        "BattSystemIdle"=dword:0A  ; 10 seconds
        "BattSuspend"=dword:12c    ; 300 seconds
```

4. Perform a clean build of the MX32ADS platform code.
   - Right click on Favorites...MX32ADS PLATFORM and select "Clean Before Building".
   - Right click again on Favorites...MX32ADS PLATFORM and select "Build Current Project".
5. Build an incremental OS image by selecting Build OS...Build and Sysgen Current BSP
6. Configure the MX32ADS board for passive KITL to (reduces overhead associated with KITL servicing).

7. Download the image to the MX32ADS. Lowest power will be observed after display timeout.

Chapter 13
Fast Infrared Driver

The Fast infrared (FIR) module is used for infrared transaction at high speeds. This module is capable of half-duplex IrDA communication. Currently, under the Windows CE implementation of the IrDA stack, we find there can be only one line of communication between two IR devices. If the source or the destination device has to be changed, then the connection has to be dropped and a fresh connection made.

IrDA supports three standards, Slow IR (SIR), Medium IR (MIR) and Fast IR (FIR). The Slow IR standard specifies a maximum speed of 115200 bps and the maximum size of a data packet is 2KB. The Fast IR supports a maximum speed of 4 Mbps. The processing requirements of FIR call for a dedicated FIR transceiver.

13.1 FIR Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX31
MXARM11 CSP Driver Path	..\ CSP\ARM\FREESCALE\MXARM11\DRIVERS\FIR
CSP Driver Path	..\ CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\FIR
CSP Static Library	<*TGTSOC*>_irfir.lib, mxarm11_irfir.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\FIR
Import Library	ndis.lib
Driver DLL	irfir.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> Infrared Communication –> FIR
SYSGEN Dependency	SYSGEN_IRDA=1, SYSGEN_OBEX_INBOX=1
BSP Environment Variables	BSP_FIR=1, BSP_NOSIR=1

13.2 Requirements

The FIR driver should meet the following requirements:

1. The driver shall conform to the Microsoft Network Driver Interface Specification (NDIS) Architecture version 5.0 in Windows CE.
2. The driver shall support IrDA protocols for infrared communication.

3. The driver shall support two power management modes, full on and full off.

13.3 Hardware Operation

Refer to the chapter on the Fast Infrared Interface (FIRI) in the hardware specification document for detailed operation and programming information.

13.3.1 FIR Interface

The FIR driver supports the Infrared Data Association (IrDA) protocol which is widely used standard for IR communication. The NDIS implementation in Windows CE includes IrDA support. The Windows CE-based IrDA protocol driver can only bind to one miniport driver at a time. In the Windows communications architecture, infrared sockets are exposed through extensions to the Winsock interface. The protocol driver provides a lower level interface (specified by NDIS) to receive incoming packets from the Miniport driver. The miniport driver assembles packets, from the raw data received from the lower level and sends these packets to the NDIS wrapper component. It also disassembles the packets received from the NDIS wrapper component and then sends the raw data to the lower level.

13.3.2 Conflicts with other SoC peripherals

13.3.2.1 i.MX31 Peripheral Conflicts

- FIRI conflicts with UART2 module. Pins TXD2, RXD2, CTS2 and RTS2 should be configured in Hardware Mode for FIR
- Since we need to support SIR along with FIR, pins TXD2 and RXD2 of UART2 have to be configured in Functional Mode for SIR

13.3.2.2 i.MX32ADS Peripheral Conflicts

13.4 Software Operation

The FIRI driver follows the Microsoft-recommended architecture for NDIS miniport drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver –> Windows CE Drivers –> Network Drivers –> Network Driver Development Concepts –> Miniports, Intermediate Drivers and Protocol Drivers**.

13.4.1 DMA Support

The FIRI driver uses the SDMA controller to transfer the IrDA packets between the IrDA protocol stack and the FIRI FIFOs. This minimizes the processing that is required by the ARM core and can also reduce the power consumption during FIRI transfer operations.

Note, however, that the FIRI driver always requires the use of DMA support for proper operation. The FIRI driver has support for an alternative non-DMA or polling-based operating mode, but this can be used only

for debugging purposes (For IrDA discovery and very small data transfers). Therefore, the BSP_SDMA_SUPPORT_FIRI macro in the bsp_cfg.h header file must always be defined as TRUE for proper operation of FIRI.

For data reception/transmission, FIRI driver uses 8bit width access for SDMA Read/Write operations respectively.

The FIRI driver configuration settings should not be changed without a detailed understanding of the platform's hardware configuration and operating characteristics. Selecting invalid or incorrect configuration settings may result in an FIRI driver that will not work properly. Conversely, the FIRI driver performance and resource usage can be fine tuned by adjusting these configuration settings.

The available compile-time DMA-related configuration options are described in Table 13-1.

Configuration Setting Name	Description
FIRI_DMA_TX_WATERMARK, FIRI_DMA_RX_WATERMARK	The transmitter and receiver watermarks that are to be used with FIRI FIFO. The default is 16 for both watermark levels.
FIRI_MAX_TX_DESC_COUNT, FIRI_MAX_RX_DESC_COUNT	Defines the number of DMA buffer descriptors. Currently, FIRI driver does not use scatter-gather list but a statically allocated DMA buffer, so only one buffer descriptor is used for transmit and receive respectively. Default is 1.
MAX_IRDA_DATA_SIZE	The maximum size of an rRDA packet. This is eventually the size of the DMA buffer used for either transmit or receive FIRI operation.

Table 13-1. FIRI Driver Configuration Options (settings.h)

This section will describe the FIRI driver DMA implementation issues and trade-off.

In order to use DMA transfers, all of the following items must be properly allocated, managed, and de-allocated by the device driver:

- The DMA data buffers where the FIRI data packets are kept.
- The DMA buffer descriptors which are used by the DMA hardware to manage the state of each DMA buffer.

The DMA data buffers is allocated from "external memory" (which is provided by off-chip external DRAM). There is flexibility in selecting the size of DMA buffer because typically the size of external memory is large without the need to worry about the possible impact and memory requirements of any other device driver. Memory allocation is handled using standard WinCE system calls.The external memory cannot be placed in low power mode while DMA is active.

The DMA buffer descriptors can also be allocated from either internal or external memory. However, in this case, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). Please refer to the CSPDDK documentation for additional information about the DDKSdmaAllocChain() API.

13.4.2 Power Management

IrDA miniport is required to export an additional mechanism for power management. This requirement allows the IrDA miniport to power down when there are no infrared sockets open. Infrared sockets are exposed through extensions to the Winsock application interface. The following list describes the expected operation of an IrDA miniport:

- The miniport is initialized without hardware resources.
- When the first infrared socket is opened, the miniport receives the OID_IRDA_REACQUIRE_HW_RESOURCES message through its set information handler. The miniport then acquires all hardware and software resources for the miniport to operate.
- When the last infrared socket is closed, the miniport receives the OID_IRDA_RELEASE_HW_RESOURCES message through its query information handler. The miniport then releases all hardware resources.

We support only full power on and full power off and hence when the NDIS queries for the OID_PNP_CAPABILITIES, we advertise power states NdisDeviceStateD0 and NdisDeviceStateD3 only.

In addition to the above, for limiting power consumption in the FIR module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. The clock gating is turned on when an socket is opened, and clock gating turned off when the socket is closed.

13.4.3 FIR Registry Settings

The following registry keys are required to properly load the FIR driver.

```
IF BSP_FIR
; Since CSPDDK has to load and configure properly
; we are shifting the order of NDIS to 2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\NDIS]
   "Order"=dword:2
[HKEY_LOCAL_MACHINE\Comm\IrDA\Linkage]
    "Bind"=multi_sz:"Irfir1"
[HKEY_LOCAL_MACHINE\Comm\Irfir]
   "DisplayName"="<TGTSOC> FIR Driver"
   "Group"="NDIS"
   "ImagePath"="irfir.dll"
[HKEY_LOCAL_MACHINE\Comm\Irfir\Linkage]
   "Route"=multi_sz:"Irfir1"
[HKEY_LOCAL_MACHINE\Comm\Irfir1\Parms]
   "BusNumber"=dword:0
   "BusType"=dword:0
   ; This is to avoid deadlock situation in NDIS
   ; when the system goes to a low power mode
   "DisablePowerManagement"=dword:1
ENDIF ; BSP_FIR
```

13.5 Unit Test

The FIR driver is tested using the IR Port Test (Winsock 1.1 and Winsock 2.0). These tests are included as part of the Windows CE 5.0 Test Kit (CETK). These test cases are used to test the functionality of a driver

for an IR Device using Winsock 1.1 and Winsock 2.0. These test cases are based on client server architecture and hence we need a pair of IR devices. This requires the server to be started before starting the client.

13.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Two installed IR devices	You must install one IR device on a test server and another IR device on a test client. You should position the IR devices in close proximity to one another.

13.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
Irapi11.dll	Test .dll file for IR Port Test (Winsock 1.1)
Irapi22.dll	Test .dll file for IR Port Test (Winsock 2.0)
Irapisrv11.exe	Test .exe file for starting server for IR Port Test (Winsock 1.1)
Irapisrv22.exe	Test .exe file for starting server for IR Port Test (Winsock 2.0)

13.5.3 Building the IR Port Tests

13.5.3.1 IR Port Test (Winsock 1.1)

The IR Port Test (Winsock 1.1) come pre-built as part of the CETK. No steps are required to build these tests. The irapi11.dll and irapisrv11.exe can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

13.5.3.2 IR Port Test (Winsock 2.0)

The IR Port Test (Winsock 2.0) come pre-built as part of the CETK. No steps are required to build these tests. The irapi22.dll and irapisrv22.exe can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

13.5.4 Running the FIR Tests

The IR Port Test (Winsock 2.0) is designed to test the functionality a driver for an infrared (IR) device using Winsock 2.0. The test is a client-server test that requires a pair of IR devices.

Note: We need to run the command, **services stop obx0:** before running server and client on both the boards.

13.5.4.1 IR Port Test (Winsock 1.1)

We can start the server in one device by typing ***irapisrv11*** at the command line. Then client side test executes on the second device using ***tux –o –d irapi11.dll*** in the command line.

For detailed information on the FIR Port tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> IR Port Test (Winsock 1.1) in the Platform Builder Help.

13.5.4.2 IR Port Test (Winsock 2.0)

We can start the server in one device by typing **irapisrv22** at the command line. Then client side test executes on the second device by using ***tux –o –d irapi22.dll*** in the command line.

For detailed information on the FIR Port tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> IR Port Test (Winsock 2.0) in the Platform Builder Help.

13.6 FIR Driver API Reference

Windows CE supports IRSock, which is a more efficient mode of IR communications. FIR device is exposed through IrDA protocol stack and we need to use IRSock for building application that makes use of IrDA. It requires a special address family named AF_IRDA. Moreover, an IRSock can only be a stream socket and not a datagram socket. The IRSock requires a header file named AF_IRDA.h for its operations.

The FIRI driver conforms to NDIS 5.0 specification by Microsoft for the miniport network drivers.

Chapter 14
General Purpose Timer (GPT) Driver

The General purpose timer is a multipurpose module used to measure intervals or generate periodic output. The timer counter value can be captured in a register using an event on an external pin. The GPT can also generate an event on a chip boundary signal and an interrupt when the timer reaches a programmed value. For i.MX31 & i.MX32, there is only one general purpose timer supported.

14.1 GPT Driver Summary

14.1.1 GPT Driver Summary for i.MX31, i.MX32

The following table provides a summary of source code location, library dependencies and other BSP information

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\GPT
CSP Driver Path	N/A
CSP Static Library	mxarm11_gpt.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\GPT
Import Library	N/A
Driver DLL	gpt.dll
Catalog Item	Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −>GPT
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPT=1

14.2 Requirements

The GPT driver should meet the following requirements:

1. The driver shall support one 32 bit up counter with clock source selection, including external clock.
2. The driver shall support the reset and free-run modes for counter operation.
3. The driver shall support two power management modes, full on and full off.
4. The driver shall be a stream interface driver implementing the programming interface defined in this document

14.3 Hardware Operation

Refer to the chapter on General Purpose Timer in the hardware specification document for detailed hardware operation and programming information.

14.3.1 Conflicts with other SoC peripherals

14.3.1.1 i.MX31, i.MX32 Peripheral Conflicts

No conflicts.

14.4 Software Operation

NOTE1

For i.MX31 and MX32, if Platform Builder profiling support is to be used, the GPT driver can not be included in the workspace.

14.4.1 Communicating with the GPT

The GPT driver controls the General Purpose Timer. This timer is used to provide high resolution (microsecond) timing functionality to other platform modules. The GPT is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the GPT, a handle to the device must first be obtained using the **GptOpenHandle** function. Subsequent commands to the device are issued using various APIs supported by this driver.

14.4.2 Creating a Handle to the GPT

14.4.2.1 i.MX31

To communicate with the GPT, a handle to the device must first be created using the **GptOpenHandle** API. The default GPT port is 1.

To open a Handle to the GPT

```
// Global data
// Handle to the GPT device
HANDLE g_hGpt = NULL;

// opening the default GPT port.
g_hGpt = GptOpenHandle();
```

For more information on this API, please see the **GptOpenHandle** section under the GPT API reference.

14.4.3 Configuring the GPT

Configuring the GPT for communications involves starting the timer and enabling the timer event trigger by calling **GptStart** API.

Before this action can be taken, a handle to the GPT port must already be opened. Call the **GptStart** API to enable and start the timer.

```
// configuring and starting the GPT
GptStart(g_hGpt) ;
```

For more information on this API, please see the **GptStart** section under the GPT API reference.

14.4.4 Write Operations

The Write operations for the GPT involve setting the time through the **GptSetTimer** API.

Before this action can be taken, a handle to the GPT must already be opened.

The timer mode can be either, *timerModeFreeRunning* or *timerModePeriodic*.

```
// Name to create the named event for Timer
#define GPT_EVENT_NAME  L"GptTest1"
// GPT Timer packet
GPT_TIMER_SET_PKT gptTimerDelayPkt;

// create an event for the timer interrupt
hGptIntr = GptCreateTimerEvent(hGpt, GPT_EVENT_NAME);

gptTimerDelayPkt.timerMode = timerModePeriodic;
gptTimerDelayPkt.period = 10000000;

// Setting the GPT timer
GptSetTimer(g_hGpt, &gptTimerDelayPkt);
```

For more information on this API, please see **GptSetTimer** section of the GPT API reference.

14.4.5 Closing the Handle to the GPT

To close the GPT handle, we need to call the **GptCloseHandle** API. But before performing the close operation, we need to stop the timer using **GptStop** API. It is always advised to call **GptReleaseTimerEvent** to release any pending timer events before closing the handle.

Before these actions can be taken, a handle to the GPT must already be opened.

To close the GPT Handle,

```
// Name to create the named event for Timer
#define GPT_EVENT_NAME  L"GptTest1"

// releasing the Timer Event.
GptReleaseTimerEvent(g_hGpt, eventString);
GptStop(g_hGpt)
GptCloseHandle(g_hGpt);
```

For more information on these APIs, please see the **GptReleaseTimerEvent**, **GptStop** and **GptCloseHandle** section under the GPT API reference.

14.4.6 Power Management

The primary method for limiting power consumption in the GPT module is to gate off all clocks to the module when GPT is not used. The clock is enabled when an application calls **GPT_Open( )**. This clock then remains enabled as long device is kept open. The GPT clock is turned off when the application closes the device using **GPT_Close()**.

14.4.6.1 PowerUp

This function will restore the state of the GPT clocks back to the state before entering suspend. If the GPT was counting before suspend, GPT will continue to count from the place where it was stopped.

14.4.6.2 PowerDown

This function will disable the clock to the GPT module. If the GPT was counting, then the count value freezes at the point when the clock is removed.

14.4.6.3 IOCTL_POWER_CAPABILITIES

N/A

14.4.6.4 IOCTL_POWER_SET

N/A

14.4.6.5 IOCTL_POWER_GET

N/A

14.4.7 GPT Registry Settings

14.4.7.1 GPT Registry Settings for i.MX31

The following registry keys are required for GPT:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\GPT]
  "Prefix"="GPT"
  "Dll"="gpt.dll"
  "Index"=dword:1
```

14.5 Unit Test

The GPT tests verify that the GPT driver properly initializes and controls the General purpose timer.

14.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
No additional hardware required	

14.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
GPTTEST.dll	Test .dll file

14.5.3 Building the GPT Tests

In order to build the GPT tests, complete the following steps:

Build an OS image for the desired configuration

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the GPT Tests directory. (For i.MX31, the directory is \WINCE500\SUPPORT\TESTS\GPT)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the GPTTEST.dll file will be located in the $(_FLATRELEASEDIR) directory.

14.5.4 Running the GPT Tests

The command line for running the GPT tests is ***tux –o –d gpttest***. The GPT tests do not contain any test specific command line options.

The following table describes the test cases contained in the GPT tests.

Test Case	Description
1: Test 1	Attempts to start the GPT timer without setting the timer period.
2: Test 2	Tests GPT timer.

14.6 GPT Driver API Reference

14.6.1 GPT Driver Functions

14.6.1.1 GptOpenHandle

14.6.1.1.1 GptOpenHandle for i.MX31

This API creates a handle to the GPT stream driver:

```
HANDLE GptOpenHandle(
void
);
```

Parameters

This API accepts no parameters.

Return Values

An open handle to the specified file indicates success. INVALID_HANDLE_VALUE indicates failure.

Remarks

Use the **GptCloseHandle** function to close the handle returned by **GptOpenHandle()**.

14.6.1.2 GptCreateTimerEvent

This API is used to create the GPT Timer event.

```
HANDLE GptCreateTimerEvent(
            HANDLE hGpt,
            LPTSTR eventName
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

eventName

[in] Pointer to a null-terminated string that specifies the name of the object.

Return Values

A non-null handle to the specified event indicates success. NULL indicates failure.

Remarks

Use the **GptReleaseTimerEvent** function to close the event. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

14.6.1.3 GptSetTimer

This API sets the timer period for the GPT.

```
BOOL GptSetTimer(
    HANDLE hGpt,
    PGPT_TIMER_SET_PKT pGptTimerDelayPkt
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

pGptTimerDelayPkt

[in] An object of the **GPT_TIMER_SET_PKT** structure.

Return Values

TRUE on success and FALSE indicates a failure.

Remarks

Mode member in *pGptTimerDelayPkt* structure can be set to one of the timer modes *timerModeFreeRunning* or *timerModePeriodic*.

14.6.1.4 GptStart

This API enables the GPT interrupt and starts the GPT timer

```
BOOL GptStart(
        HANDLE hGpt
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

Return Values

TRUE on success and FALSE indicates a failure.

Remarks

None.

14.6.1.5 GptStop

This API disables the GPT interrupt and stops the GPT timer.

```
BOOL GptStop(
```

```
        HANDLE hGpt
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

Return Values

TRUE on success and FALSE indicates a failure.

Remarks

None.

14.6.1.6 GptReleaseTimerEvent

This API closes the currently open GPT Timer Event.

```
BOOL GptReleaseTimerEvent(
        HANDLE hGpt,
        LPTSTR eventName
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

eventName

[in] Pointer to a null-terminated string that specifies the name of the object

Return Values

Nonzero indicates success.
Zero indicates failure.
To get extended error information, call **GetLastError()**.

Remarks

None.

14.6.1.7 GptCloseHandle

This API closes a handle to the GPT driver.

```
BOOL GptCloseHandle(
        HANDLE hGpt
);
```

Parameters

hGpt

[in] Handle to the GPT driver returned by **GptOpenHandle** API.

Return Values

Nonzero indicates success.
Zero indicates failure.
To get extended error information, call **GetLastError().**

Remarks

None.

14.6.2 GPT Driver Structures

14.6.2.1 GPT_TIMER_SET_PKT

14.6.2.1.1 GPT_TIMER_SET_PKT for i.MX31

```
typedef struct
{
timerMode_c timerMode;
UINT32 period;
} GPT_TIMER_SET_PKT, *PGPT_TIMER_SET_PKT;
```

Members

timerMode

*S*elects between two supported modes: reset or periodic mode (*timerModePeriodic*) and free-running mode (*timerModeFreeRunning*).

period

Counter period (in milli-seconds)

Chapter 15
Hantro Codecs Drivers

The Hantro MPEG4 Video VGA Encoder is an IP wrapper for the 5250 Hardware VGA MPEG4/H.263 Encoder. The Hantro MPEG4/H.263 Decoder is fully implemented in software. Both the Encoder and Decoder drivers are based on the Microsoft DirectShow architecture.

15.1 Codecs Drivers Summary

The following table provides a summary of binary code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31
Target SOC (TGTSOC)	MX31
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\HANTRO_CODECS
Import Library	strmbase.lib, msdmo.lib, strmiids.lib, dmoguids.lib, coredll.lib, quartz.lib, ole32.lib, oleaut32.lib, uuid.lib, mmtimer.lib
Driver DLL/LIB	htr_pp.lib, htrdecdmo.dll, htrdecdmo.lib, htrencftr.dll, htrencftr.lib, mp4enc.dll, mp4enc.lib, sw_decoder.lib, htrdecfltr.dll, htrdecfltr.lib
Catalog Item	Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −> Hantro_Codecs
SYSGEN Dependency	SYSGEN_DSHOW_DMO=1 SYSGEN_DSHOW=1 SYSGEN_DSHOW_VIDREND=1
BSP Environment Variables	BSP_HANTRO_CODECS=1

15.2 Requirements

The Hantro codecs should meet the following requirements:

1. The encoder shall support MPEG-4 Simple Profile and H.263 Profile 0.
2. The encoder shall support video resolutions up to VGA (640x480) at up to 30 frames per second.
3. The encoder shall support Microsoft DirectShow filter. (Note: On MX31, DirectShow encoder filter is implemented)

4. The decoder shall support MPEG4 Simple Profile levels from 0 to 5 and H.263 Profile 0 levels from 10 to 45.
5. The decoder shall support resolutions up to VGA (640x480) at up to 30 frames per second.
6. The decoder shall support Microsoft DirectShow DMO (DirectX Media Objects).

15.3 Hardware Operation

Refer to the chapter on the Hantro Encoder in the hardware specification document for detailed operation and programming information.

15.3.1 Conflicts with other SoC peripherals

15.3.1.1 i.MX31 Peripheral Conflicts

Hantro codecs do not have conflicts with any other module.

15.4 Software Operation

The codecs drivers follow the Microsoft-recommended architecture for DirectShow. The details of this architecture and its operation can be found in the Platform Builder Help or www.microsoft.com.

15.4.1 Power Management

Power management is not implemented in the Hantro codecs drivers.

15.4.2 Codecs Registry Settings

The following registry keys are required to properly load the decoder drivers:

```
[HKEY_CLASSES_ROOT\CLSID\{6AD7CB74-75B7-4887-86AE-90099A10C5E2}]
@="Hantro Decoder DMO"
[HKEY_CLASSES_ROOT\CLSID\{6AD7CB74-75B7-4887-86AE-90099A10C5E2}\InprocServer32]
@="htrdecdmo.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\DirectShow\MediaObjects\6ad7cb74-75b7-4887-86ae-90099a10c5e2]
@="Hantro Decoder DMO"
"InputTypes"=hex:76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,4d,50,34,56,\
  00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,10,00,80,00,00,aa,00,\
  38,9b,71,6d,70,34,76,00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,\
  10,00,80,00,00,aa,00,38,9b,71,48,32,36,33,00,00,10,00,80,00,00,aa,00,38,9b,\
  71,76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,68,32,36,33,00,00,10,00,\
  80,00,00,aa,00,38,9b,71
"OutputTypes"=hex:76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,49,59,55,56,\
  00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,10,00,80,00,00,aa,00,\
  38,9b,71,49,59,55,56,00,00,10,00,80,00,00,aa,00,38,9b,71

[HKEY_CLASSES_ROOT\DirectShow\MediaObjects\Categories\4a69b442-28be-4991-969c-b500adf5d8a8\6a
d7cb74-75b7-4887-86ae-90099a10c5e2]
```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{6AD7CB74-75B7-4887-86AE-90099A10C5E2}]
@="Hantro Decoder DMO"
[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{6AD7CB74-75B7-4887-86AE-90099A10C5E2}\InprocServe
r32]
@="htrdecdmo.dll"
"ThreadingModel"="Both"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\DirectShow\MediaObjects\6ad7cb74-75b7-4887-86ae-90099a10
c5e2]
@="Hantro Decoder DMO"
"InputTypes"=hex:76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,4d,50,34,56,\
  00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,10,00,80,00,00,aa,00,\
  38,9b,71,6d,70,34,76,00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,\
  10,00,80,00,00,aa,00,38,9b,71,48,32,36,33,00,00,10,00,80,00,00,aa,00,38,9b,\
  71,76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,68,32,36,33,00,00,10,00,\
  80,00,00,aa,00,38,9b,71
"OutputTypes"=hex:76,69,64,73,00,00,10,00,80,00,00,aa,00,38,9b,71,49,59,55,56,\
  00,00,10,00,80,00,00,aa,00,38,9b,71,76,69,64,73,00,00,10,00,80,00,00,aa,00,\
  38,9b,71,49,59,55,56,00,00,10,00,80,00,00,aa,00,38,9b,71

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\DirectShow\MediaObjects\Categories\4a69b442-28be-4991-96
9c-b500adf5d8a8\6ad7cb74-75b7-4887-86ae-90099a10c5e2]
```

The following registry keys are required to properly load the encoder drivers:

```
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}]
@="Hantro MPEG-4/H.263 Video Encoder Filter"
"Merit"=dword:00600000
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\InprocServer32]
@="htrencftr.dll"
"ThreadingModel"="Both"

[HKEY_CLASSES_ROOT\Filter\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}]
@="Hantro MPEG-4/H.263 Video Encoder Filter"

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input]
"Direction"=dword:00000000
"IsRendered"=dword:00000000
"AllowedZero"=dword:00000000
"AllowedMany"=dword:00000000
"ConnectsToPin"="Output"
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{56555949-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{56555949-0000-0010-8000-00AA00389B71}\]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output]
"Direction"=dword:00000001
"IsRendered"=dword:00000000
```

```
"AllowedZero"=dword:00000000
"AllowedMany"=dword:00000000
"ConnectsToPin"="Input"
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{5634504D-0000-0010-8000-00AA00389B71}]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{5634504D-0000-0010-8000-00AA00389B71}\]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{7634706D-0000-0010-8000-00AA00389B71}]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{7634706D-0000-0010-8000-00AA00389B71}\]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363248-0000-0010-8000-00AA00389B71}]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363248-0000-0010-8000-00AA00389B71}\]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363268-0000-0010-8000-00AA00389B71}]

[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363268-0000-0010-8000-00AA00389B71}\]

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Mp4Enc]
   "Prefix"="MPE"
   "Dll"="mp4enc.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"

; Hantro MPEG-4/H.263 Video Decoder Filter

; DLL registration
[HKEY_CLASSES_ROOT\CLSID\{9A81E196-3BBA-4821-B18B-21BB496F80F8}]
@="Hantro MPEG-4/H.263 Video Decoder Filter"
"Merit"=dword:00600000
[HKEY_CLASSES_ROOT\CLSID\{9A81E196-3BBA-4821-B18B-21BB496F80F8}\InprocServer32]
@="htrdecfltr.dll"
"ThreadingModel"="Both"

; Registration for DirectShow filtergraph
[HKEY_CLASSES_ROOT\Filter\{9A81E196-3BBA-4821-B18B-21BB496F80F8}]
@="Hantro MPEG-4/H.263 Video Decoder Filter"

; Input pin
; Direction      = 0 [Input]
; Rendered       = 0 [Not rendered]
```

```
; AllowedZero= 0   [Required to be connected always when running]
; AllowedMany= 0   [Many instances NOT allowed]
; Output types  = MEDIATYPE_Video; MP4V, mp4v, H263, h263
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output]
"Direction"=dword:00000000
"IsRendered"=dword:00000000
"AllowedZero"=dword:00000000
"AllowedMany"=dword:00000000
"ConnectsToPin"="Input"
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{5634504D-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{5634504D-0000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{7634706D-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{7634706D-0000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363248-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363248-0000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363268-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Output\Types\{73646976-0
000-0010-8000-00AA00389B71}\{33363268-0000-0010-8000-00AA00389B71}\]

; Pin descriptions
[HKEY_CLASSES_ROOT\CLSID\{9A81E196-3BBA-4821-B18B-21BB496F80F8}\Pins]

; Output pin
; Direction      = 1 [Output]
; Rendered       = 0 [Not rendered]
; AllowedZero= 0   [Required to be connected always when running]
; AllowedMany= 0   [Many instances NOT allowed]
; Input types   = MEDIATYPE_Video; MEDIASUBTYPE_IYUV, MEDIASUBTYPE_YV12
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input]
"Direction"=dword:00000000
"IsRendered"=dword:00000000
"AllowedZero"=dword:00000000
"AllowedMany"=dword:00000000
"ConnectsToPin"="Output"
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{56555949-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{56555949-0000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{32315659-0000-0010-8000-00AA00389B71}]
```

```
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{32315659-0000-0010-8000-00AA00389B71}\]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{E436EB7B-0000-0010-8000-00AA00389B71}]
[HKEY_CLASSES_ROOT\CLSID\{9FD1B4C9-BF1E-4E7D-ACE6-586BBB3280C6}\Pins\Input\Types\{73646976-00
00-0010-8000-00AA00389B71}\{E436EB7B-0000-0010-8000-00AA00389B71}\]
```

15.5 Unit Test

For the Hantro codecs unit test, we provide simple test bench source code and test data, which are located in \WINCE500\SUPPORT\TESTS\HANTRO_CODECS or \WINCE600\SUPPORT\MX31\TESTS \HANTRO_CODECS. The test bench would be an example for developing DirectShow applications.

In order to reduce the BSP package size, a limited amount of test data has been included. Additional test data may be developed by the user. See “Customizing the Test Bench/Data” section for detailed instructions.

15.5.1 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	Description
htrenctb.exe	Test bench for the encoder testing for MX31 with the default test data.
htrdecdmotb.exe	Test bench for the decoder testing for MX31 with the default test data.

15.5.2 Building the Codecs Test Bench

15.5.2.1 Test Bench in Windows CE

To build the codecs test bench in Windows CE, complete the following steps.

Build an OS image for the desired configuration:

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option on Windows CE 5.0, or go to the **Build** menu option and select the **Open Release Directory in Build Window** menu option on Windows CE 6.0. This will open a command prompt.
- Change to the HANTRO_CODECS tests directory (\WINCE500\SUPPORT\TESTS\HANTRO_CODECS or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS)
- Enter “*set wincerel=1*” on the command prompt and hit <return>. This will copy the generated “.exe” to the flat release directory.
- To build the encoder test bench for MX31, change to \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\ENCODE\htrenctb or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\ENCODE\htrenctb,
- Enter “*build –c*” at the prompt, and then press <return>. The htrenctb.exe file will be located in the $(_FLATRELEASEDIR) directory.

- Copy \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\ENCODE\testdata or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\ENCODE\testdata to the $(_FLATRELEASEDIR) directory.
- To build the decoder DMO test bench, change to \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\DECODE\htrdectb or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\DECODE\htrdectb,
- Enter "*build –c*" at the prompt, and then press <return>. The htrdecdmotb.exe file will be located in the $(_FLATRELEASEDIR) directory.
- Copy \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\DECODE\test_sequences or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\DECODE\test_sequences to the $(_FLATRELEASEDIR) directory.

15.5.3 Customizing the Test Bench/Data

To include your customized test files for encoding:

- Place the data files and reference stream files at: \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\ENCODE\testdata or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\ENCODE\testdata.
- Modify the TestCases.h file for MX31 at \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\ENCODE\htrenctb\TestCases.h or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\ENCODE\htrenctb\TestCases.h file.
- Follow the steps in the previous section for building encoder test bench.

NOTE

In TestSettings.h, we set "true" to "keepOutputFiles" and "false" to "runComparation" as defaults. You can change them as what you want.

To include your customized test files for decoding:

- Place the data files and reference stream files at: \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\DECODE\test_sequences or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\DECODE\test_sequences.
- Modify the file[] and referenceData[] variables at \WINCE500\SUPPORT\TESTS\HANTRO_CODECS\DECODE\htrdectb\ DecoderFilterTestBench.cpp or \WINCE600\SUPPORT\MX31\TESTS\HANTRO_CODECS\DECODE\htrdectb\ DecoderFilterTestBench.cpp.
- Follow the steps in the previous section for building decoder test bench.

NOTE

For the decoder DMO test bench, we uncommented #define VIDEO_RENDERER in DecoderFilterTestBench.cpp, so that we can use DirectShow provided Video Renderer as a default. If you want to use file based testing, just comment #define VIDEO_RENDERER in DecoderFilterTestBench.cpp.

15.5.4 Running the Codecs Tests

After downloading an OS image to the board, the test bench can be executed from the Windows CE target shell with one of the following commands:

- Encoder filter test for MX31: *"s htrenctb.exe"*
- Decoder DMO test for MX31: "*s htrdecdmotb.exe"*

15.6 Codecs Drivers API Reference

DirectShow detailed reference information may be found in Platform Builder Help or www.microsoft.com.

For the Hantro codecs engine APIs information, please refer to "5250_API_User_Manual.pdf" and "MPEG4_Decoder_API_User_Manual.pdf", located in \WINCE500\SUPPORT\TESTS\HANTRO _CODECS\DOCS or \WINCE600\SUPPORT\MX31\TESTS\HANTRO _CODECS\DOCS of this BSP release package.

Chapter 16
Inter-Integrated Circuit (I2C) Driver

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

16.1 I2C Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

16.1.1 For MX31, MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\I2C
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTPLAT*>\DRIVERS\I2C
CSP Static Libraries	mxarm11_i2c.lib, <*TGTSOC*>_i2c.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\I2C
Import Library	N/A
Driver DLL	i2c.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> I2C Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_I2CBUS=1

16.2 Requirements

The I2C driver should meet the following requirements:

1. The driver shall support the I2C communication protocol.
2. The driver shall support multiple I2C controllers.
3. The driver shall support the I2C master mode of operation.
4. The driver shall not support the I2C slave mode of operation.
5. The driver shall be a stream interface driver implementing the programming interface defined in this document.
6. The driver shall support two power management modes, full on and full off.

16.3 Hardware Operation

Refer to the chapter on I2C in the hardware specification document for detailed operation and programming information.

16.3.1 Conflicts with other SoC peripherals

16.3.1.1 i.MX31 and i.MX32 Peripheral Conflicts

The i.MX31 and i.MX32 platforms contain three I2C modules, but only one of these modules, I2C1, may be used without any conflicts on the ADS board. I2C2 does not have any allocated pins, and I2C3 shares pins with the CSPI2 module. The CSPI2 signals are selected in the IOMUX, as they are required for proper communication with the MC13783 PMIC.

In order to use I2C1 to communicate with the onboard USB transceiver, jumpers must be placed correctly on ADS board jumpers JP6 and JP7 to connect pins 2 and 3. This configuration is required for correct USB operation as well as for passing the CETK test, which communicates with the USB transceiver.

16.4 Software Operation

16.4.1 Communicating with the I2C

The I2C is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the I2C, 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 basic steps are detailed below.

16.4.2 Creating a Handle to the I2C

Call the **CreateFile** function to open a connection to the I2C device. An I2C port must be specified in this call. The format is “I2CX”, with X being the number indicating the I2C port. This number should not exceed the number of I2C instances on the platform. 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: as the I2C port.
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.

The following code example shows how to open an I2C port.

```
// Open the serial port.
hI2C = CreateFile (CAM_I2C_PORT,  // 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.

16.4.3 Configuring the I2C

Configuring the I2C port for communications involves 2 main operations:

- Setting the I2C frequency
- Setting the Self Address (the address for the I2C port on the platform).

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.

To configure an I2C port:

1. Set the *hDevice* parameter to the previously acquired I2C port handle.
2. Set the *dwIoControlCode* to one of the following IOCTL codes:

- I2C_IOCTL_SET_FREQUENCY
- I2C_IOCTL_SET_SELF_ADDR

3. Set the *lpInBuffer* to point to the variable that you are wishing to use for the I2C port setting. Set nInBufferSize to the size of that variable.
4. Set *lpOutBuffer*, *lpBytesReturned*, and *lpOverlapped* to NULL. Set *nOutBufferSize* to 0.

The following code example shows how to configure the I2C port.

```
// Clock frequency set at 1MHz
DWORD dwFrequency = 1000000;         // I2C frequency
BYTE bySelf = 0x20;                  // Self address value

// Set I2C frequency
DeviceIoControl(hI2C,                       // file handle to the driver
                I2C_IOCTL_SET_FREQUENCY, // I/O control code
                (PBYTE) &dwFrequency,    // in buffer
                sizeof(dwFrequency),     // in buffer size
                NULL,                    // out buffer
                0,                       // out buffer size
                NULL,                    // number of bytes returned
                NULL);                   // ignored (=NULL)

// Set I2C self address
DeviceIoControl(hI2C,                       // file handle to the driver
                I2C_IOCTL_SET_SELF_ADDR, // I/O control code
```

```
(PBYTE) &bySelf,          // in buffer
sizeof(bySelf),           // in buffer size
NULL,                     // out buffer
0,                        // out buffer size
NULL,                     // number of bytes returned
NULL);                    // ignored (=NULL)
```

As a substitute for the **DeviceIoControl** calls above, macros may be used to simplify the code. The following are examples:

```
I2C_MACRO_SET_FREQUENCY(hI2C, dwFrequency);
I2C_MACRO_SET_SELF_ADDR(hI2C, bySelf);
```

16.4.4 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. 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.

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) Set the *byAddr* field to the 7-bit I2C slave address of the device to which the data will be written.

      NOTE

      The byAddr field requires the 7-bit I2C slave address, aligned to the least significant 7 bits. This address will be shifted left one bit and ORed with the read/write bit to compose the 8-bit value sent out during the I2C slave address cycle. In older versions of this driver, the slave address was entered as the most significant 7 bits of the 8-bit value.

   c) 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 is set to I2C_RW_READ, create a buffer of bytes to hold the data which will be read from the slave device.
   d) Set the *wLen* field to size, in bytes, of the read or write buffer. This will indicate the number of bytes to write or read.
   e) Set the *lpiResult* field to point to an integer that will hold the return value from the write operation.
2. Set the *hDevice* parameter to the previously acquired I2C port handle.
3. Set the *dwIoControlCode* to the I2C_IOCTL_TRANSFER IOCTL code.

4. Set the *lpInBuffer* to point to the I2C_TRANSFER_BLOCK 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.
6. After calling the **DeviceIoControl** function, check the *lpiResult* field to ensure that the operation was successful. If *lpiResult* points to the I2C_NO_ERROR value, the operation was successful. Otherwise, there was an error.

The following code example demonstrates how to perform a transfer that contains one write and one read packet. The write is performed before the read operation.

```
    I2C_TRANSFER_BLOCK I2CXferBlock;
    I2C_PACKET I2CPacket[2];
    BYTE byAddr = 0x2D;    // Slave Address
    BYTE byOutData = 0x39; // Data to write
    BYTE byInData;         // Read buffer

    // Packet 0 contains write operation
    I2CPacket[0].pbyBuf = (PBYTE) &byOutData;
    I2CPacket[0].wLen = sizeof(byOutData);

    I2CPacket[0].byRW = I2C_RW_WRITE;
    I2CPacket[0].byAddr = byAddr;
    I2CPacket[0].lpiResult = lpiResult;

// Packet 1 contains read operation
    I2CPacket[1].pbyBuf = (PBYTE) &byInData;
    I2CPacket[1].wLen = sizeof(byInData);

    I2CPacket[1].byRW = I2C_RW_READ;
    I2CPacket[1].byAddr = byAddr;
    I2CPacket[1].lpiResult = lpiResult;

    I2CXferBlock.pI2CPackets = I2CPacket;
    I2CXferBlock.iNumPackets = 2;

    // Transfer data via I2C
    DeviceIoControl(hI2C,                   // file handle to the driver
                    I2C_IOCTL_WRITEREG,  // I/O control code
                    (PBYTE) &I2CXferBlock,  // in buffer
                    sizeof(I2CXferBlock),   // 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, macros may be used to simplify the code. The following is an example:

```
    I2C_MACRO_TRANSFER(hI2C, &I2CXferBlock);
```

Repeated Start

The array of I2C_PACKET objects passed to the Transfer command is guaranteed to be performed sequentially, without interruption or preemption by another driver that is attempting to access the I2C module. An I2C START command initiates the transmission of the first packet in the I2C_TRANSFER_BLOCK array. For subsequent packets, a change in the direction of communication (from Read to Write or Write to Read) or a change in the target slave address triggers a REPEATED START command before the transmission of the packet. Thus, if a REPEATED START is required between data transfers with a target I2C device, all of those data transfers should be contained within a single I2C_TRANSFER_BLOCK. The final packet in the I2C_TRANSFER_BLOCK is succeeded by an I2C STOP command.

16.4.5 Closing the Handle to the I2C

Call the **CloseHandle** function to close a handle to the I2C when an application is done using it.

CloseHandle has one parameter, which is the handle returned by the **CreateFile** function call that opened the I2C port.

There is a two-second delay after **CloseHandle** is called before the port is closed and resources are freed. This delay allows pending operations to complete.

16.4.6 Power Management

The primary method for limiting power consumption in the I2C module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. In the Windows CE 5.0 <TGTPLAT> BSP, the I2C module always operates in master mode and never in slave mode. As a result, the I2C module can be disabled, and its clocks turned off, whenever the module is not processing I2C packets. By contrast, were the I2C module to operate in slave mode, the module would have to be enabled, and have its clocks turned on, at all times in order to properly receive the interrupt that signals the start of a data transfer from another I2C master device.

As described in the **Data Transfer Operations** section, I2C data transfer operations are handled in I2C_TRANSFER_BLOCK objects, which contain one or more packets of I2C data. The I2C driver turns on the I2C clocks and enables the I2C module before processing an I2C_TRANSFER_BLOCK, and then disables and turns off clocks to the I2C module after the block of packets has been processed. This limits the time during which the I2C module is consuming power to the time during which the I2C is actively performing data transfers.

16.4.6.1 PowerUp

This function is not implemented for the I2C driver. Power to the I2C module is managed as I2C transfer operations are processed. There are no additional power management steps needed for the I2C.

16.4.6.2 PowerDown

This function is not implemented for the I2C driver.

16.4.6.3 IOCTL_POWER_SET

This function is not implemented for the I2C driver.

16.4.7 I2C Registry Settings

The following registry keys are required to properly load the I2C1 module.

```
For MX31 and MX32:
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C1]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:1
   "Order"=dword:19
```

16.5 Unit Test

The I2C tests verify that the I2C driver properly initializes and controls the MX I2C controller.

16.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
No additional hardware required on MX31, MX32	

16.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
i2ctest.dll	Test .dll file

Before running the tests, the camera must be initialized by the software to be able to respond to I2C commands. This initialization includes starting the IPU/CSI clocks, enabling the CSI and reseting the camera sensor.

16.5.3 Building the I2C Tests

In order to build the I2C tests, complete the following steps:

Build an OS image for the desired configuration

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the I2C Tests directory. (MX31 and MX32: \WINCE500\SUPPORT\TESTS\I2C)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the i2ctest.dll file will be located in the $(_FLATRELEASEDIR) directory.

16.5.4 Running the I2C Tests

The command line for running the I2C tests is ***tux –o –d i2ctest***. The I2C tests do not contain any test specific command line options.

The following table describes the test cases contained in the I2C tests.

Test Case	Description
100: I2C Soft Reset Test	Verifies that the I2C is reset properly.
1000: I2C Read/Write Test	Attempts a series of reads and writes to registers in the USB onboard transceiver (on MX31, MX32)

16.6 I2C Driver API Reference

16.6.1 I2C Driver IOCTLS

This section consists of descriptions for the I2C I/O control codes (IOCTLs). These IOCTLs are used in calls to **DeviceIoControl** to issue commands to the I2C device. Only relevant parameters for the IOCTL have a description provided.

16.6.1.1 I2C_IOCTL_GET_CLOCK_RATE

This **DeviceIoControl** request retrieves the clock rate divisor. Note that the value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I2C specifications to obtain the true frequency.

Parameters

lpOutBuffer Pointer to the divisor index. The true clock frequency is platform dependent. Refer to I2C specification for more information.

nOutBufferSize Size in bytes of the divisor index.

16.6.1.2 I2C_IOCTL_GET_READ_ADDR

This **DeviceIoControl** request retrieves the current target I2C device read address. This address should be set before attempting an I2C_IOCTL_RECEIVE or I2C_IOCTL_READREG command.

Parameters

lpOutBuffer Pointer to the current target I2C device read address. The valid range of the address is [0x00, 0x7F].

nOutBufferSize Size in bytes of the read address.

16.6.1.3 I2C_IOCTL_GET_SELF_ADDR

This **DeviceIoControl** request retrieves the address of the I2C device. Note that this macro is only meaningful if it is currently in Slave mode.

Parameters

lpOutBuffer Pointer to the current I2C device address. The valid range of the address is [0x00, 0x7F].

nOutBufferSize Size in bytes of the I2C device address.

16.6.1.4 I2C_IOCTL_IS_MASTER

This **DeviceIoControl** request determines whether the I2C is currently in Master mode.

Parameters

lpOutBuffer Pointer to a BYTE that will contain 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. This should be one byte.

16.6.1.5 I2C_IOCTL_IS_SLAVE

This **DeviceIoControl** request determines whether the I2C is currently in Slave mode.

Parameters

lpOutBuffer Pointer to a BYTE that will contain 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. This should be one byte.

16.6.1.6 I2C_IOCTL_RESET

This **DeviceIoControl** request performs a hardware reset. Note that the I2C driver will still maintain all of the current information of the device, including all of the initialized addresses.

16.6.1.7 I2C_IOCTL_SET_CLOCK_RATE

This **DeviceIoControl** request initializes the I2C device with the given clock rate. Note that this IOCTL does not expect to receive the absolute peripheral clock frequency. Rather, it will be expecting the clock rate divisor index stated in the I2C specification. If absolute clock frequency must be used, please use the macro I2C_MACRO_SET_FREQUENCY.

Parameters

lpInBuffer Pointer to the divisor index. Refer to I2C specification to obtain the true clock frequency.

nInBufferSize Size in bytes of the divisor index.

16.6.1.8 I2C_IOCTL_SET_FREQUENCY

This **DeviceIoControl** request estimates the nearest clock rate acceptable for I2C device and initialize the I2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, please refer to the macro I2C_MACRO_GET_CLOCK_RATE to determine the estimated index.

Parameters

lpInBuffer Pointer to the desired I2C frequency.

nInBufferSize Size in bytes of the I2C frequency requested.

16.6.1.9 I2C_IOCTL_SET_MASTER_MODE

This **DeviceIoControl** request sets the I2C device to Master mode.

16.6.1.10 I2C_IOCTL_SET_SELF_ADDR

This **DeviceIoControl** request initializes the I2C device with the given address.

Parameters

lpInBuffer Pointer to the expected I2C device address. The valid range of addresses is [0x00, 0x7F].

nInBufferSize Size in bytes of the I2C device address.

Remarks The device will be expected to respond when any master on the I2C bus wishes to proceed with any transfer. Note that this IOCTL will have no effect if the I2C device is in Master mode.

16.6.1.11 I2C_IOCTL_SET_SLAVE_MODE

This **DeviceIoControl** request sets the I2C device to Slave mode.

16.6.1.12 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.

16.6.2 I2C Driver Macros

16.6.2.1 I2C_MACRO_GET_CLOCK_RATE

This macro will retrieve the clock rate divisor.

```
I2C_MACRO_GET_CLOCK_RATE(
  HANDLE hDev,
  WORD wClkRate
);
```

Parameters

hDev	The I2C device handle retrieved from CreateFile().
wClkRate	Contains the divisor index. Refer to I2C specification to obtain the true clock frequency.
Return Values	Returns TRUE or FALSE. If the result is TRUE, the operation is successful.
Remarks	Note that the value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I2C specification to obtain the true frequency.

16.6.2.2 I2C_MACRO_GET_SELF_ADDR

This macro will retrieve the current I2C device address. Note that this macro is only meaningful if it is currently in Slave mode.

```
I2C_MACRO_GET_SELF_ADDR(
  HANDLE hDev,
  WORD bySelfAddr
);
```

Parameters

hDev	The I2C device handle retrieved from CreateFile().
dwSelfAddr	The current I2C 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.

16.6.3 I2C_MACRO_IS_MASTER

This macro determines whether the I2C is currently in Master mode.

```
I2C_MACRO_IS_MASTER(
  HANDLE hDev,
  BOOL bIsMaster
);
```

Parameters

hDev	The I2C device handle retrieved from CreateFile().

bIsMaster TRUE if the I2C device is in Master mode.

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

16.6.3.1 I2C_MACRO_IS_SLAVE

This macro determines whether the I2C is currently in Slave mode.

```
I2C_MACRO_IS_SLAVE(
  HANDLE hDev,
  BOOL bIsSlave
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

bIsSlave TRUE if the I2C device is in Slave mode.

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

16.6.3.2 I2C_MACRO_RESET

This macro perform a hardware reset. Note that the I2C driver will still maintain all of the current information of the device, including the initialized addresses.

```
I2C_MACRO_RESET(
  HANDLE hDev,
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

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

16.6.3.3 I2C_MACRO_SET_CLOCK_RATE

This macro will initialize the I2C device with the given clock rate.

```
I2C_MACRO_SET_CLOCK_RATE(
  HANDLE hDev,
  WORD wClkRate
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

wClkRate Contains the divisor index. Refer to I2C specification to obtain the true clock frequency.

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

Remarks Note that this macro does not expect to receive the absolute peripheral clock frequency. Rather, it will be expecting the clock rate divisor index stated in the I2C specification. If absolute clock frequency must be used, please use the macro I2C_MACRO_SET_FREQUENCY.

16.6.3.4 I2C_MACRO_SET_FREQUENCY

This macro will estimate the nearest clock rate acceptable for I2C device and initialize the I2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, please refer to the macro I2C_MACRO_GET_CLOCK_RATE to determine the estimated index.

```
I2C_MACRO_SET_FREQUENCY (
  HANDLE hDev,
  DWORD dwFreq
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

dwFreq The desired frequency.

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

16.6.3.5 I2C_MACRO_SET_MASTER_MODE

This macro set the I2C device to Master mode.

```
I2C_MACRO_SET_MASTER_MODE(
  HANDLE hDev,
  BOOL bIsMaster
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

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

16.6.3.6 I2C_MACRO_SET_SELF_ADDR

This macro initializes the I2C device with the given address.

```
I2C_MACRO_SET_SELF_ADDR(
  HANDLE hDev,
  BYTE bySelfAddr
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

bySelfAddr The expected I2C device address. The valid range for the address is [0x00, 0x7F].

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

Remarks The device will be expected to respond when any master on the I2C bus wishes to proceed with any transfer. Note that this macro will have no effect if the I2C device is in Master mode.

16.6.3.7 I2C_MACRO_SET_SLAVE_MODE

This macro set the I2C device to Slave mode.

```
I2C_MACRO_SET_SLAVE_MODE(
  HANDLE hDev,
  BOOL bIsSlave
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

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

16.6.3.8 I2C_MACRO_TRANSFER

This macro performs a sequence of data transfers to a target device. All of the required information should be stored in the I2C_TRANSFER_BLOCK object passed in the *pI2CTransferBlock* field.

```
I2C_MACRO_TRANSMIT(
  HANDLE hDev,
  PI2C_TRANSFER_BLOCK pI2CTransferBlock
);
```

Parameters

hDev The I2C device handle retrieved from CreateFile().

pI2CTransferBlock

pI2CPackets [in] Pointer to an array of packets to be transferred sequentially.

iNumPackets [in] The 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.

16.6.4 I2C Driver Structures

16.6.4.1 I2C_PACKET

This structure contains the information needed to write or read data using an I2C port.

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

Members

byAddr This 7-bit slave address specifies the target I2C device to or from which data will be read or written.

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 A pointer to a buffer of bytes. For a Read operation, this is the buffer into which data will be read. For a Write operation, this buffer contains the data to write to the target device.

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.

16.6.4.2 I2C_TRANSFER_BLOCK

This structure contains an array of packets to be transferred using an I2C port.

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

Members

pI2CPackets A pointer to an array of I2C_PACKET objects.

iNumPackets The number of I2C_PACKET objects pointed to by *pI2CPackets*.

Chapter 17
Keypad Driver

The Keypad Port (KPP) module is used for keypad matrix scanning. This module is capable of detecting, debouncing, and decoding one or two keys pressed simultaneously in the keypad.

The keypad driver converts input from the KPP into keyboard events that the driver enters into the input system. The driver is also responsible for generating the proper Unicode characters from these keyboard events.

17.1 Keypad Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

17.1.1 MX31, MX32 Keypad Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\KEYBD
CSP Driver Path	N/A
CSP Static Library	mxarm11_Keypad.lib mxarm11_PddList.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\KEYBD
Import Library	N/A
Driver DLL	Kbdmouse.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> Input Devices –> EVB Keypad
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_KBDMOUSE_EVBKPD=1

17.2 Requirements

The Keypad driver should meet the following requirements:

1. The driver shall conform to the Microsoft Layout Manager Interface.

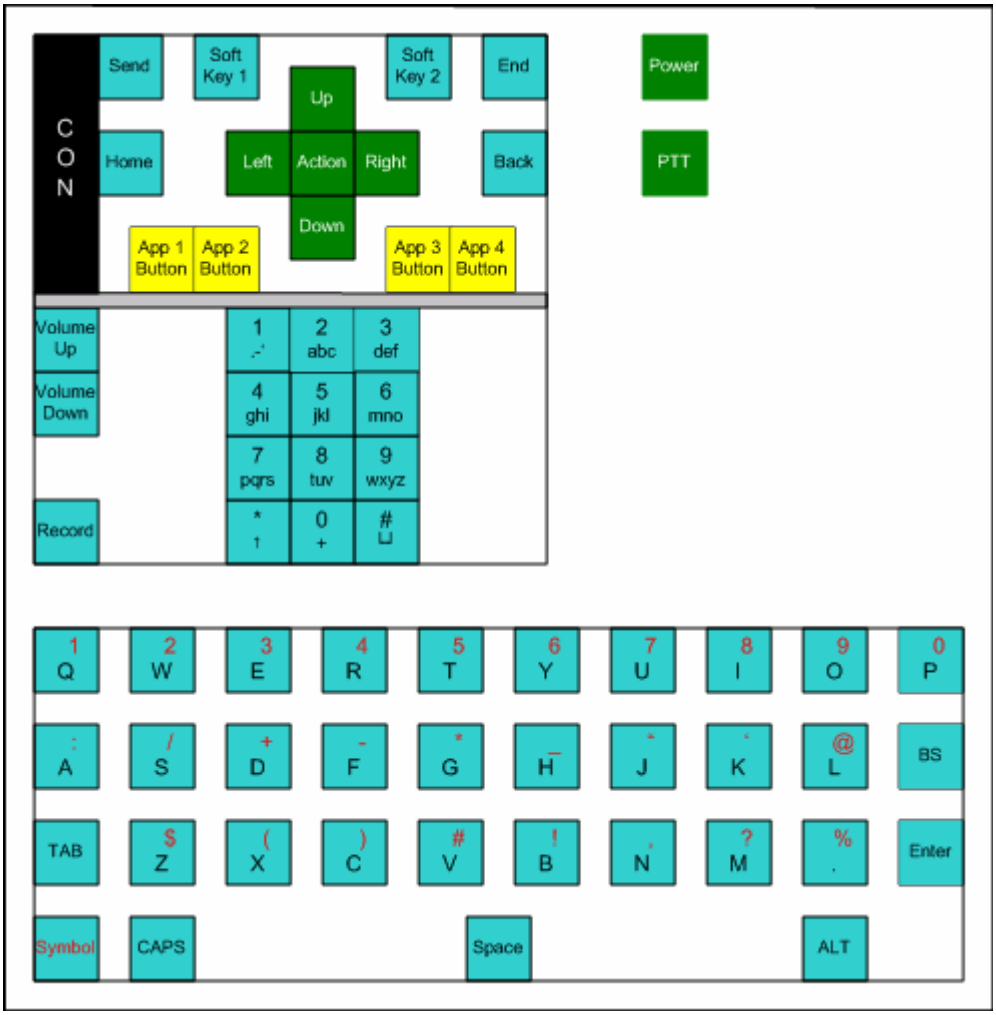
2. The driver shall support the ADS Keypad.
3. The MX31, MX32 keypad drivers shall support inputs from USB keyboard.
4. The driver shall support multiple simultaneous key presses.
5. The driver shall support two power management modes, full on and full off.

17.3 Hardware Operation

Refer to the chapter on the KPP in the hardware specification document for detailed operation and programming information.

17.3.1 The MX31 and MX32 ADS Keypad

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



The Symbol key is displayed in red at the bottom left corner of the keypad. Several keys on the keypad display two characters: one in black, and one in red. When the Symbol key is pressed simultaneously with one of these keys, the red character is entered. Otherwise, the character in black is entered.

Another key of note is the ALT key, at the bottom-right corner of the keypad. On the physical keypad, the silk screen label is ON/OFF. However, this functionality is not supported, and the key is instead used as an ALT key. The ALT key provides the user with greater ability to navigate Windows CE. The following key combinations make use of the ALT key to perform specific tasks in Windows CE:

Press	To
ALT + TAB	Switch between open items.
ALT + underlined letter in a menu name	Display the corresponding menu.
ALT + Enter	Open the properties for the selected object.

17.3.2 Conflicts with other SoC peripherals

17.3.2.1 MX31 and MX32 Peripheral Conflicts

No conflicts.

17.4 Software Operation

The Keypad driver follows the Microsoft-recommended architecture for keyboard drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver** –> **Windows CE Drivers** –> **Keyboard Drivers** –> **Keyboard Driver Development Concepts**.

17.4.1 MX31 and MX32 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 first converted to intermediate PS/2 keyboard scan code values and then converted into virtual keys, which are hardware independent numbers that identify the key. On the other hand, if a key is pressed from the keyboard, the generated scan code is directly converted into virtual keys. For alphabetic keys, the ASCII code for the capitalized letter is the virtual key. For other keys, the virtual key is defined by Microsoft and starts with “VK_”.

The following table shows the scan code to virtual key mapping for the i.MX31 and i.MX32 ADS Keypad:

Key	Keypad Scan Code	Virtual Key
SEND	9	VK_TTALK
KEY 1	1	VK_TSOFT1
KEY 2	40	VK_TSOFT2

Key	Keypad Scan Code	Virtual Key
END	48	VK_TEND
UP	32	VK_UP
LEFT	8	VK_LEFT
ACTION	0	VK_TACTION
RIGHT	24	VK_RIGHT
DOWN	16	VK_DOWN
HOME	17	VK_THOME
BACK	56	VK_TBACK
APP 1	25	VK_APP1
APP 2	41	VK_APP2
APP 3	49	VK_APP3
APP 4	57	VK_APP4
VOL UP	33	VK_TVOLUMEUP
VOL DOWN	34	VK_TVOLUMEDOWN
1	18	VK_T1
2	10	VK_T2
3	2	VK_T3
4	26	VK_T4
5	50	VK_T5
6	58	VK_T6
7	42	VK_T7
8	19	VK_T8
9	3	VK_T9
RECORD	43	VK_TRECORD
*	35	VK_TSTAR
0	27	VK_T0
#	11	VK_TPOUND
Q	51	“Q”
W	59	“W”
E	28	“E”
R	44	“R”
T	52	“T”
Y	60	“Y”
U	54	“U”

Key	Keypad Scan Code	Virtual Key
I	62	"I"
O	63	"O"
P	5	"P"
A	4	"A"
S	12	"S"
D	20	"D"
F	36	"F"
G	61	"G"
H	14	"H"
J	38	"J"
K	46	"K"
L	39	"L"
BS	4	VK_BACK
TAB	5	VK_TAB
Z	29	"Z"
X	3	"X"
C	4	"C"
V	5	"V"
B	6	"B"
N	22	"N"
M	3	"M"
.	2	K_PERIOD
ENTER	31	VK_RETURN
SYMBOL	13	VK_SYMBOL
CAPS	21	VK_CAPITAL
SPACE	7	VK_SPACE
ON/OFF	15	VK_MENU

17.4.2 Power Management

The primary method for limiting power consumption in the Keypad module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. In this module, the clocks are enabled only when it is required to access any Keypad register. Once done using the registers, the clocks are brought back to their previous state.

17.4.2.1 BSPKppPowerOn

This function is used to power up the keypad. This function will do the necessary configuration settings in the registers to bring up the keypad and then the clocks are brought back to their original state as it was just before the module was powered down.

17.4.2.2 BSPKppPowerOff

This function would power down the keypad. But before turning off the module, the current state of the clock settings to this module is saved and then waited until keypad does not report any key-down/key-up event. Then, the clocks to this module are turned off.

17.4.2.3 IOCTL_POWER_CAPABILITIES

N/A

17.4.2.4 IOCTL_POWER_SET

N/A

17.4.2.5 IOCTL_POWER_GET

N/A

17.4.3 Keypad Registry Settings

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

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\KEYBD]
    "CalVKey"=dword:0
    "ContLessVKey"=dword:0
    "ContMoreVKey"=dword:0
    "TaskManVKey"=dword:2E
    "Keyboard Type"=dword:4
    "Keyboard SubType"=dword:0
    "Keyboard Function Keys"=dword:0
    "Keyboard Layout"="00000409"
    "DriverName"="kbdmouse.dll"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Layouts\00000409]
    "Layout File"="kbdmouse.dll"
    "Layout Text"="US-Keypad"
    "KPPLayout"="kbdmouse.dll"

[HKEY_CURRENT_USER\Keyboard Layout\Preload\4]
    @="00000409"
```

17.5 Unit Test

The Keypad driver is tested using the Keyboard Test included as part of the Windows CE 5.0 Test Kit (CETK). The Keyboard Test is an interactive test that measures the functionality of a keyboard driver. The test verifies that all key sequences, chording, text editing, repeat rate, and delay are functioning properly.

17.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
MX31 or MX32 ADS Keypad for testing the MX31 or MX32 driver.	United States keypad supported by the keypad driver being tested.

17.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
kbdtest.dll	Test .dll file

17.5.3 Building the Keyboard Tests

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

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

17.5.4 Running the Keyboard Tests

The command line for running the Keyboard test is ***tux –o –d kbdtest***.

For detailed information on the Keyboard tests, see **Debugging and Testing** –> **Tools for Debugging and Testing** –> **Windows CE Test Kit** –> **CETK Tests** –> **Keyboard Test** –> **Keyboard Test Cases** in the Platform Builder Help.

As the Keyboard tests are designed to test a standard PS/2 keyboard with a standard set of keys, several of the tests will fail or should be skipped for the Keypad. The following table describes the Keyboard test cases and notes if the tests are expected to fail.

Test Case	Description
50: Manual key press	Displays keyboard events on the screen as they occur. This allows you to verify that the keyboard events are correctly recognized. Press the spacebar as indicated in the test to end the test. This test case fails only if you choose **No** at the end of the test.
51: Key sequence check	Tests the ability to detect key input properly. This test requires you to press keys on the keyboard. Press each key in the order specified when prompted. This test case fails if you input an incorrect keystroke and then choose **No** when prompted to run the test a second time. **Note:** This test passes for all keys on the ADS Keypad, and fails for all keys not on the ADS Keypad.
52: Key chording	Tests the ability to detect certain key combinations properly. This test prompts you to press keys on the keyboard in a specified order. This test fails if you input incorrect keystrokes and then choose **No** when prompted to the test a second time. You must press and hold down keys that are part of a key combination. For example, when prompted to press <CTRL-ALT-A>, press and hold the CTRL key, press and hold the ALT key, and then press the A key. You can then release all keys. **Note:** This test will fail, as all key chords evaluated in the test require unsupported keys (CTRL, SHIFT, etc.)
53: Text Editing	Provides a text box that allows you to test all keys manually. When this test is finished, choose the box near the top of the screen to continue. This test fails only if you choose **No** at the end of the test.
54: Repeat rate and key delay	Tests the ability to speed up and slow down the key repeat rate and the ability to increase and decrease the delay before repeat. The test requires you to observe and remember the repeat rate and key delay and to respond to questions accordingly. This test fails only if you choose **No** when prompted.
55: Async key test	Tests that specific keys on the keyboard are detected properly. This test prompts you to press keys on the keyboard in a specified order. This test fails if you input an incorrect keystroke and then choose **No** when prompted to the test a second time. **Note:** This test passes for all keys on the ADS Keypad, and fails for all keys not on the ADS Keypad.

17.6 Keypad Driver API Reference

Detailed reference information for the Keypad driver may be found in Platform Builder Help at the following location:

Developing a Device Driver –> Windows CE Drivers –> Keyboard Drivers –> Keyboard Driver Reference

17.6.1 Keypad PDD Functions

The following table shows a mapping of Keyboard PDD functions to the functions used in the Keypad driver:

PDD Function Pointer	Keypad Driver Function
PFN_KEYBD_PDD_ENTRY	KPP_Entry
PFN_KEYBD_PDD_GET_KEYBD_EVENT	KeybdPdd_GetEventEx2
PFN_KEYBD_PDD_POWER_HANDLER	KPP_PowerHandler

Chapter 18
MBX Direct3D Mobile/OpenGL ES Drivers

The MBX Lite graphics processor is an IP wrapper for 2D/3D hardware acceleration, and is designed for ultra-low-power cost-sensitive system-on-chip (SOC) applications such as mainstream mobile phones, PDAs and handheld gaming devices.

The MBX D3DM and OpenGL ES drivers interface with the i.MX31 Image Processing Unit (IPU) Synchronous Display Controller (SDC) to combine graphics and video planes and to generate display controls with programmable timing. This module is designed to be compatible with the Sharp LQ035Q7DB02 QVGA LCD panel and the NEC NL6448BC20 VGA LCD panel.

The MBX Direct3D Mobile (D3DM) driver provides the actual drawing services that Microsoft Direct3D Mobile middleware uses. The middleware is a thin layer of software that handles all call transport, synchronization, and OS integration issues; the driver manages all the memory for display surfaces.

OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems. OpenGL ES 1.X is for fixed function hardware and offers acceleration, image quality and performance.

The MX31 MBX supports the D3DM and OpenGL ES in Windows CE 5.0.

18.1 Direct3D Mobile/OpenGL ES Drivers Summary

The following table provides a summary of binary code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31
Target SOC (TGTSOC)	MX31
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\MBX
Import Library	ddgpe.lib, gpe.lib
Driver DLL	Clcdckmif.dll, sdc_display.dll, ddi_powervr.dll, gxdma.dll, libGLES_CM.dll, libpvrWCEWSEGL.dll, IMGEGL.dll, pvr_d3dm.dll, pvr_kernel.dll, um3partyif.dll

Driver Attribute	Definition
Catalog Item	Third Party BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> Display –> NEC NL6448BC20 (VGA) or Sharp LQ035Q7DB02 (QVGA) Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –>MBX –>MBX MX31 Base Driver Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –>MBX –>MBX MX31 D3DM Core Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –>MBX –>MBX MX31 Ogles Core
SYSGEN Dependency	SYSGEN_DDRAW=1 SYSGEN_D3DM=1
BSP Environment Variables	BSP_MBX BSP_DISPLAY_NEC_NL6448BC20 = 1 for NEC VGA Panel BSP_DISPLAY_SHARP_LQ035Q7DB02 = 1 for Sharp LCD Panel

18.2 Requirements

The D3DM and OpenGL ES drivers should meet the following requirements:

1. The driver shall derive from the DirectDraw Graphics Primitive Engine (DDGPE) class.
2. The driver shall support the DirectDraw Hardware Abstraction Layer (DDHAL).
3. The driver shall support the Sharp LQ035Q7DB02 QVGA and NEC NL6448BC20 VGA panels.
4. Direct3D Mobile shall support Microsoft Direct3D Mobile Specification.
5. OpenGL ES shall support OpenGL ES 1.1 Specification.

18.3 Hardware Operation

Refer to the chapter on the MBX in the hardware specification document for detailed operation and programming information.

18.3.1 Conflicts with other SoC peripherals

18.3.1.1 i.MX31 Peripheral Conflicts

MBX does not have conflicts with any other module.

18.4 Software Operation

The MBX D3DM driver follows the Microsoft-recommended architecture for Direct3D Mobile drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver –> Windows CE Drivers –> Direct3D Mobile Display Drivers.**

18.4.1 Communicating with the MBX

Communications with the MBX drivers are accomplished through Microsoft-defined APIs or OpenGL ES APIs.

The MBX Direct3D Mobile driver uses the local hooking model. The application's process space includes both the Microsoft® Direct3D® Mobile middleware (d3dm.dll) and the MBX Direct3D Mobile driver. Because the locally hooked drivers are not loaded into the Graphics, Windowing, and Event Subsystem (GWES), they do not have direct access to the hardware. As a result, these drivers must rely on some other graphics technology, such as DirectDraw or the Graphics Device Interface (GDI), to present rendered output to the user.

18.5 Configuring the LCD Display Panels

The display is configured based on the **PanelType** registry key, which is described in the **Display Registry Settings** section below. The **PanelType** registry key indicates the display panel that is being used. There are currently two supported display panels: the Sharp LQ035Q7DB02 QVGA LCD panel and NEC NL6448BC20 VGA LCD panel.

18.5.1 LCD Display Registry Settings

The following registry keys are optionally included, depending on the display panel catalog item included in the OS design. If the Sharp QVGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
      "Bpp"=dword:10                ; 16bpp
      "VideoBpp"=dword:10           ; RGB565
      "PanelType"=dword:0           ; Sharp QVGA Panel
      "VideoMemSize"=dword:350000   ; 3.5MB
```

If the NEC VGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
      "Bpp"=dword:10                ; 16bpp
      "VideoBpp"=dword:10           ; RGB565
      "PanelType"=dword:1           ; NEC VGA Panel
      "VideoMemSize"=dword:350000   ; 3.5MB
```

18.5.2 Power Management

Power management is not implemented in the D3DM and OpenGL ES drivers yet.

18.5.3 Direct3D Mobile and OpenGL ES Registry Settings

The following registry keys are required to properly load the MBX D3DM and OpenGL ES drivers.

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Timeouts]
      "ACUserIdle"=dword:00000000
      "ACSystemIdle"=dword:00000000
      "ACSuspend"=dword:00000000
      "BattUserIdle"=dword:00000000
      "BattSystemIdle"=dword:00000000
      "BattSuspend"=dword:00000000

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PVRKernel]
      "Prefix"="PKM"
```

```
      "Dll"="pvr_kernel.dll"
      "Order"=dword:1
      "Keep"=dword:1
      ; Indicate PKM is a generic power manageable interface
      "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PVR3rdPartyKernel]
      "Prefix"="P3P"
      "Dll"="clcdckmif.dll"
      "Order"=dword:2
      "Keep"=dword:1
      ; Indicate P3P is a generic power manageable interface
      "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"

[HKEY_CURRENT_USER\ControlPanel\Keybd]
      "Contrast"=dword:80

[HKEY_LOCAL_MACHINE\SYSTEM\GWE]
      "PORepaint"=dword:3

[HKEY_LOCAL_MACHINE\System\GDI\Drivers]
      "Display"="ddi_powervr.dll"

[HKEY_LOCAL_MACHINE\Drivers\Display\PowerVR]
      "IsrDll"="GIISR.DLL"
      "IsrHandler"="ISRHandler"
      "OutputMask"=dword:003f3f3f
      "DisableDynamicScreenRotation"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\Display\PowerVR]
      "HWRecoveryTimeout"="350"

[HKEY_LOCAL_MACHINE\Drivers\Display\PowerVR\MBX1\Game Settings\OpenGLES]
[HKEY_LOCAL_MACHINE\Drivers\Display\PowerVR\MBX1\Game Settings\D3DM]
[HKEY_LOCAL_MACHINE\PowerVR\MBX1\Game Settings\OpenGLES]
[HKEY_LOCAL_MACHINE\PowerVR\MBX1\Game Settings\D3DM]

[HKEY_LOCAL_MACHINE\system\gdi\rotation]
      "Angle"=dword:0

[HKEY_LOCAL_MACHINE\System\D3DM\Drivers]
      "LocalHook"="pvr_d3dm.dll"
```

18.6 Unit Test

To add all MBX-related drivers on Windows CE to the image, including D3DM, OpenGL ES, DirectDraw/GDI, LCD, and IPU SDC, the three MBX modules should be added from the catalog: *MBX MX31 Base driver*, *MBX MX31 D3DM Core*, and *MBX MX31 Ogles Core*.

The following sections will focus on D3DM Windows CETK test as well as D3DM/OpenGL ES demos test. For more details on DirectDraw/GDI CETK and Windows Media Player tests, please refer to the documentation section on the Display Driver.

18.6.1 Unit Test Hardware

The following table lists the required hardware to run the GDI, DirectDraw, D3DM and OpenGL ES tests:

Requirements	Description
SHARP LQ035Q7DB02 QVGA Panel or NEC NL6448BC20 VGA Panel	Display panel required for display of graphics data.

18.6.2 Unit Test Software

18.6.2.1 Direct3D Mobile Interface Tests

The following table lists the required software to run the D3DM Interface tests.

Requirements	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
D3DM_Interface.dll	Dynamic-link library for the test.

For the Direct3D Mobile Interface Test to run, in your OS design, set the SYSGEN_D3DM variable from **Catalog** –> **Core OS** –>**Windows CE devices** –> **Graphics** –>**Direct3D Mobile**. You must also include a Direct3D Mobile driver in your OS design.

18.6.2.2 Direct3D Mobile Driver Verification Tests

The following table lists the software required to run the D3DM Driver Verification tests:

Requirements	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
D3DM_DriverVerif.dll	Dynamic-link library for the test.
QAD3DMX.dll	Library that supplies functions to compute three-dimensional mathematical operations required by the test.

For the Direct3D Mobile Driver Verification Test to run, in your OS design, set the SYSGEN_D3DM variable from **Catalog** –> **Core OS** –>**Windows CE devices** –> **Graphics** –>**Direct3D Mobile** and SYSGEN_D3DMREF variable from **Catalog** –> **Device Drivers** –> **Direct3D Mobile** –> **Direct3D Mobile Reference Driver.** You must also include a Direct3D Mobile driver in your OS design.

18.6.2.3 Direct3D Mobile Driver Comparison Tests

The following table lists the software required to run the D3DM Driver Comparison tests:

Requirements	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
D3DM_DriverComp.dll	Dynamic-link library for the test.
D3DMImageManagement.dll	Library that provides utilities for capturing, comparing, and storing rendered output.
QAD3DMX.dll	Library that supplies functions to compute three-dimensional mathematical operations required by the test.

For the Direct3D Mobile Driver Comparison Test to run, in your OS design, set the SYSGEN_D3DM variable from **Catalog** –> **Core OS** –>**Windows CE devices** –> **Graphics** –>**Direct3D Mobile** and SYSGEN_D3DMREF variable from **Catalog** –> **Device Drivers** –> **Direct3D Mobile** –> **Direct3D Mobile Reference Driver.** You must also include a Direct3D Mobile driver in your OS design.

18.6.3 Building the Direct3D Mobile Tests

The D3DM Interface Tests, D3DM Driver Verification Tests and D3DM Driver Comparison Tests come pre-built as part of the CETK. No steps are required to build these tests. The D3DM_Interface.dll, D3DM_DriverVerif.dll, D3DM_DriverComp.dll, D3DMImageManagement.dll, QAD3DMX.dll files can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

18.6.4 Running the Direct3D Mobile Tests

18.6.4.1 Running the Direct3D Mobile Interface Tests

The command line for running the D3DM Interface tests is:

tux –o –d d3dm_interface.dll

For detailed information on the D3DM Interface tests and command line options for these tests, see **Debugging and Testing** –> **Tools for Debugging and Testing** –> **Windows CE Test Kit** –> **CETK Tests** –> **Direct3D Mobile Interface Test** in the Platform Builder Help.

The following table describes the test cases contained in the D3DM Interface test suite:

Test Case	Description
1-99	Tests the methods for the IDirect3DMobile interface.
101-199	Tests the methods for the IDirect3DMobileDevice interface.

Test Case	Description
201-299	Tests the methods for the IDirect3DMobileIndexBuffer interface.
301-399	Tests the methods for the IDirect3DMobileSurface interface.
401-499	Tests the methods for the IDirect3DMobileTexture interface.
501-599	Tests the methods for the IDirect3DMobileVertexBuffer interface.
2001-2099	Tests the security of a Direct3D Mobile driver.

18.6.4.2 Running the Direct3D Mobile Driver Verification Tests

The command line for running the D3DM Driver Verification tests is:

tux –o –d d3dm_DriverVerif.dll

For detailed information on the D3DM Interface tests and command line options for these tests, see **Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Direct3D Mobile Driver Verification Test** in the Platform Builder Help.

The following table describes the test cases contained in the D3DM Driver Verification test suite:

Test Case	Description
1-199	Tests the IDirect3DMobileDevice::ProcessVertices method. These test cases verify transformation and lighting, primarily covering permutations of the following factors: D3DMRS_AMBIENT with various settings D3DMRS_AMBIENTMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL D3DMRS_COLORVERTEX: FALSE , TRUE D3DMRS_DIFFUSEMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL D3DMRS_LIGHTING: FALSE , TRUE D3DMRS_LOCALVIEWER: FALSE , TRUE D3DMRS_SPECULARENABLE: FALSE, TRUE D3DMRS_SPECULARMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL D3DMTRANSFORMSTATETYPE settings, including various D3DMMATRIX values for D3DMTS_VIEW, D3DMTS_WORLD, and D3DMTS_PROJECTION Various D3DMVIEWPORT settings for the IDirect3DMobileDevice::SetTransform method Various D3DMLIGHT settings for the IDirect3DMobileDevice::SetLight method
201-899	Tests blending. These test cases render scenes and verify output, primarily covering permutations of the following factors: D3DMRS_SRCBLEND and D3DMRS_DESTBLEND: D3DMBLEND_ZERO D3DMBLEND_ONE D3DMBLEND_SRCCOLOR D3DMBLEND_INVSRCCOLOR D3DMBLEND_SRCALPHA D3DMBLEND_INVSRCALPHA D3DMBLEND_DESTALPHA D3DMBLEND_INVDESTALPHA D3DMBLEND_DESTCOLOR D3DMBLEND_INVDESTCOLOR D3DMBLEND_SRCALPHASAT D3DMRS_BLENDOP D3DMBLENDOP_ADD D3DMBLENDOP_SUBTRACT D3DMBLENDOP_REVSUBTRACT D3DMBLENDOP_MIN D3DMBLENDOP_MAX Flexible vertex format (FVF) component values, including various color values

Test Case	Description
900-1099	Tests stencils. These test cases render scenes and verify output, primarily covering permutations of the following factors: Various values of D3DMRS_STENCILMASK D3DMRS_STENCILFAIL and D3DMRS_STENCILPASS D3DMSTENCILOP_KEEP D3DMSTENCILOP_ZERO D3DMSTENCILOP_REPLACE D3DMSTENCILOP_INCRSAT D3DMSTENCILOP_DECRSAT D3DMSTENCILOP_INVERT D3DMSTENCILOP_INCR D3DMSTENCILOP_DECR D3DMRS_STENCILFUNC D3DMCMP_GREATER D3DMCMP_LESS Various values of D3DMRS_STENCILREF
1100-1199	Tests resource management. These test cases exercise the managed resource functionality of a driver. These test cases create resources until video memory constraints require that one or more resources be evicted. These test cases assess resources of the following types: D3DMRTYPE_INDEXBUFFER D3DMRTYPE_VERTEXBUFFER D3DMRTYPE_TEXTURE

18.6.4.3 Running the Direct3D Mobile Driver Comparison Tests

The command line for running the D3DM Driver Comparison tests is:

tux –o –d d3dm_DriverComp.dll

For detailed information on the D3DM Interface tests and command line options for these tests, see **Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Direct3D Mobile Driver Comparison Test** in the Platform Builder Help.

The following table describes the test cases contained in the D3DM Driver Comparison test suite:

Test Case	Description
0-99	Tests culling. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_CULLMODE: D3DMCULL_CW D3DMCULL_CCW D3DMCULL_NONE Primitive orientations: Back facing Front facing Edge on
100-199	Tests lighting. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_AMBIENT, with a variety of settings D3DMRS_AMBIENTMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL D3DMRS_COLORVERTEX: FALSE, TRUE D3DMRS_DIFFUSEMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL D3DMRS_LIGHTING: FALSE, TRUE D3DMRS_LOCALVIEWER: FALSE, TRUE D3DMRS_SPECULARENABLE: FALSE, TRUE D3DMRS_SPECULARMATERIALSOURCE: D3DMMCS_COLOR1 D3DMMCS_COLOR2 D3DMMCS_MATERIAL Flexible vertex format (FVF) values including various positions, normals, and colors Various D3DMMATERIAL values for the IDirect3DMobileDevice::SetMaterial method Various D3DMLIGHT values for the IDirect3DMobileDevice::SetLight method Various D3DMMATRIX values for the IDirect3DMobileDevice::SetTransform method
200-299	Tests depth buffers. These test cases render scenes that primarily cover permutations of the following factors: Various D3DMCLEAR_ZBUFFER depth values for the IDirect3DMobileDevice::Clear method D3DMPRIMITIVETYPE: D3DMPT_POINTLIST D3DMPT_LINELIST D3DMPT_TRIANGLELIST D3DMRS_ZENABLE: D3DMZB_FALSE, D3DMZB_TRUE D3DMRS_ZFUNC: D3DMCMP_ALWAYS D3DMCMP_NEVER D3DMCMP_LESS D3DMCMP_EQUAL D3DMCMP_LESSEQUAL D3DMCMP_GREATER D3DMCMP_NOTEQUAL D3DMCMP_GREATEREQUAL D3DMRS_ZWRITEENABLE: FALSE, TRUE Vertex positions to exercise various depths

Test Case	Description
300-399	Tests primitive rasterization. These test cases render scenes that primarily cover permutations of the following factors: D3DMPRIMITIVETYPE: D3DMPT_POINTLIST D3DMPT_LINELIST D3DMPT_LINESTRIP D3DMPT_TRIANGLELIST D3DMPT_TRIANGLESTRIP D3DMPT_TRIANGLEFAN D3DMRS_COLORWRITEENABLE: With and without D3DMCOLORWRITEENABLE_RED With and without D3DMCOLORWRITEENABLE_GREEN With and without D3DMCOLORWRITEENABLE_BLUE D3DMRS_FILLMODE: D3DMFILL_SOLID, D3DMFILL_POINT, D3DMFILL_WIREFRAME FVF number format: D3DMFMT_D3DMVALUE_FIXED, D3DMFMT_D3DMVALUE_FLOAT Primitive drawing functions: **DrawPrimitive, DrawIndexedPrimitive** Primitive drawing range: Various *StartVertex* and *PrimitiveCount* values for the **DrawPrimitive** function, both including cases that use all vertices within the active IDirect3DMobileVertexBuffer, and including cases that use only a subset of the vertices within the active **IDirect3DMobileVertexBuffer** Various *BaseVertexIndex*, *minIndex*, *NumVertices*, *startIndex*, and *PrimCount* values for the **DrawIndexedPrimitive** function, including cases that use all vertices and indices within the active **IDirect3DMobileVertexBuffer** or IDirect3DMobileIndexBuffer, and cases that use only a subset of the vertices within the active **IDirect3DMobileVertexBuffer** or **IDirect3DMobileIndexBuffer**
400-599	Tests clipping. These test cases render scenes that primarily cover permutations of the following factors: D3DMPRIMITIVETYPE: D3DMPT_LINELIST, D3DMPT_TRIANGLELIST D3DMRS_FILLMODE: D3DMFILL_POINT, D3DMFILL_SOLID, D3DMFILL_WIREFRAME D3DMRS_SHADEMODE: D3DMSHADE_GOURAUD, D3DMSHADE_FLAT FVF components with D3DMFVF_DIFFUSE, and with or without D3DMFVF_SPECULAR Primitive position relative to the view frustum: Primitives that are wholly outside of the view frustum Primitives that are wholly inside of the view frustum Primitives that are partially inside and partially outside of the view frustum
600-699	Tests FVF. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_SHADEMODE: D3DMSHADE_GOURAUD, D3DMSHADE_FLAT D3DMTSS_TEXTURETRANSFORMFLAGS: With and without D3DMTTFF_PROJECTED With and without D3DMTTFF_COUNT2 With and without D3DMTTFF_COUNT3 FVF components: With and without D3DMFVF_DIFFUSE With and without D3DMFVF_SPECULAR With and without the texture coordinate counts D3DMFVF_TEX1, D3DMFVF_TEX2, D3DMFVF_TEX3, and D3DMFVF_TEX4 With and without the texture coordinate sizes D3DMFVF_TEXCOORDSIZE1(0), D3DMFVF_TEXCOORDSIZE2(0), and D3DMFVF_TEXCOORDSIZE3(0)

Test Case	Description
700-899	Tests fogging. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_FOGCOLOR: Various D3DMCOLOR settings D3DMRS_FOGDENSITY: Various values in the [0.0f, 1.0f] range D3DMRS_FOGENABLE: TRUE, FALSE D3DMRS_FOGSTART, D3DMRS_FOGEND: For D3DMRS_FOGTABLEMODE, various values in the [0.0f, 1.0f] range For D3DMRS_FOGVERTEXMODE, various values in the [fNear, fFar] range, where fNear and fFar are view frustum planes D3DMRS_FOGTABLEMODE: D3DMFOG_NONE D3DMFOG_LINEAR D3DMFOG_EXP D3DMFOG_EXP2 D3DMRS_FOGVERTEXMODE: D3DMFOG_NONE D3DMFOG_LINEAR D3DMFOG_EXP D3DMFOG_EXP2 D3DMTRANSFORMSTATETYPE: D3DMTS_PROJECTION settings with various near and far planes
900-999	Tests mipmaps. These test cases render scenes that primarily cover permutations of the following factors: D3DMTSS_MAGFILTER: D3DMTEXF_POINT, D3DMTEXF_LINEAR, D3DMTEXF_ANISOTROPIC D3DMTSS_MINFILTER: D3DMTEXF_POINT D3DMTEXF_LINEAR D3DMTEXF_ANISOTROPIC D3DMTSS_MIPFILTER: D3DMTEXF_NONE D3DMTEXF_POINT D3DMTEXF_LINEAR D3DMTSS_MIPMAPLODBIAS: Various values in the [-2.0f, 2.0f] range Primitive vertices, which vary to produce primitive extents that can isolate specific mipmap levels
1000-2099	Tests transformations. These test cases render scenes that primarily cover permutations of the following factors: D3DMTRANSFORMSTATETYPE settings: Various rotation, shear, scale, and translation matrices for D3DMTS_VIEW Various rotation, shear, scale, and translation matrices for D3DMTS_WORLD

Test Case	Description
2100-2299	Tests StretchRect operations. These test cases render scenes that primarily cover permutations of the following factors: D3DMPOOL, for source and destination surfaces: D3DMPOOL_VIDEOMEM D3DMPOOL_SYSTEMMEM D3DMTEXTUREFILTERTYPE: D3DMTEXF_NONE D3DMTEXF_LINEAR D3DMTEXF_POINT IDirect3DMobileSurface, originating from GetBackBuffer, CreateImageSurface, and IDirect3DMobileTexture::GetSurfaceLevel Relative source and destination RECT extents: Identical extents Extents requiring shrinking Extents requiring stretching
2500-2599	Tests CopyRect operations. These test cases render scenes that primarily cover permutations of the following factors: D3DMPOOL, for source and destination surfaces: D3DMPOOL_VIDEOMEM D3DMPOOL_SYSTEMMEM **IDirect3DMobileSurface**, originating from **GetBackBuffer**, **CreateImageSurface**, and **IDirect3DMobileTexture::GetSurfaceLevel**
2700-2799	Tests ColorFill operations. These test cases render scenes that primarily cover permutations of the following factors: D3DMPOOL, for source and destination surfaces: D3DMPOOL_VIDEOMEM D3DMPOOL_SYSTEMMEM **IDirect3DMobileSurface**, originating from **GetBackBuffer**, **CreateImageSurface**, and **IDirect3DMobileTexture::GetSurfaceLevel**
2800-2899	Tests last pixels. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_LASTPIXEL: FALSE, TRUE D3DMPRIMITIVETYPE: D3DMPT_LINELIST, D3DMPT_LINESTRIP
3000-3099	Tests texture wrapping. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_WRAP0 settings: D3DMWRAPCOORD_0 D3DMWRAPCOORD_1 D3DMWRAPCOORD_0 | D3DMWRAPCOORD_1 FVF components: D3DMFVF_TEX1 with D3DMFVF_TEXCOORDSIZE1(0) D3DMFVF_TEX1 with D3DMFVF_TEXCOORDSIZE2(0) FVF component values, including various position and texture coordinate values

Test Case	Description
3100-3299	Tests alpha test operations. These test cases render scenes that primarily cover permutations of the following factors: D3DMTSS_ALPHAARG1: D3DMTA_DIFFUSE, D3DMTA_TEXTURE D3DMTSS_ALPHAARG2: D3DMTA_CURRENT, D3DMTA_DIFFUSE D3DMRS_ALPHAFUNC: D3DMCMP_ALWAYS D3DMCMP_NEVER D3DMCMP_LESS D3DMCMP_EQUAL D3DMCMP_LESSEQUAL D3DMCMP_GREATER D3DMCMP_NOTEQUAL D3DMCMP_GREATEREQUAL D3DMTSS_ALPHAOP: D3DMTOP_MODULATE, D3DMTOP_SELECTARG1 D3DMRS_ALPHAREF: Various values in the [0,255] range Contents of various surfaces
3500-3599	Tests depth bias. These test cases render scenes that primarily cover permutations of the following factors: D3DMRS_DEPTHBIAS D3DMRS_SLOPESCALEDEPTHBIAS FVF component values, including various position and diffuse color values
3600-3699	Tests swap chains. These test cases render scenes that primarily cover permutations of the following factors: Various source and destination **RECT** structures for IDirect3DMobileDevice::Present.

Test Case	Description
3700-3799	Tests drawing. These test cases render scenes that primarily cover permutations of the following factors: D3DMPRIMITIVETYPE: D3DMPT_TRIANGLELIST, D3DMPT_TRIANGLESTRIP D3DMRS_CULLMODE: D3DMCULL_CW D3DMCULL_CCW D3DMCULL_NONE Various drawing orders among primitives with identical depth values
4000-4799	Tests texture stages. These test cases render scenes that primarily cover permutations of the following factors: D3DMTEXTUREOP: D3DMTOP_SELECTARG1 D3DMTOP_SELECTARG2 D3DMTOP_MODULATE D3DMTOP_MODULATE2X D3DMTOP_MODULATE4X D3DMTOP_ADD D3DMTOP_ADDSIGNED D3DMTOP_ADDSIGNED2X D3DMTOP_SUBTRACT D3DMTOP_ADDSMOOTH D3DMTOP_BLENDDIFFUSEALPHA D3DMTOP_BLENDTEXTUREALPHA D3DMTOP_BLENDFACTORALPHA D3DMTOP_BLENDTEXTUREALPHAPM D3DMTOP_MULTIPLYADD D3DMTOP_LERP D3DMTSS_COLORARG0, D3DMTSS_ALPHAARG0, D3DMTSS_COLORARG1, D3DMTSS_ALPHAARG1, D3DMTSS_COLORARG2, D3DMTSS_ALPHAARG2: D3DMTA_TFACTOR D3DMTA_SPECULAR D3DMTA_TEXTURE D3DMTA_DIFFUSE D3DMTA_COMPLEMENT D3DMTA_ALPHAREPLICATE Various D3DMCOLOR values input as texture parameters

18.6.5 Direct3D Mobile/OpenGL ES Application Samples/Demos

In order to reduce the OS image size, a limited number of pre-built demos have been included in the BSP package.

18.6.5.1 Direct3D Mobile Application Samples

There are seven D3DM application samples located in \WINCE500\PUBLIC\DIRECTX\SDK\SAMPLES\D3DM, which will be built when BSP_MBX is enabled.

Tests	Pass
D3dm_createdevice.exe	Y
D3dm_fixedpoint.exe	Y

Tests	Pass
D3dm_lights.exe	Y
d3dm_matrices.exe	Y
d3dm_textures.exe	Y
d3dm_twotri.exe	Y
d3dm_vertices.exe	Y

18.6.5.2 Direct3D Mobile/OpenGL ES Demos

Tests	Pass
D3DMEvilSkull.exe	Y
OGLESVase.exe	Y

Please refer to the MX31 MBX SDK package for more demo applications in both source code and binary code.

18.7 Drivers API Reference

18.7.1 Direct3D Mobile

Documentation for the Direct3D driver APIs can be found within the Platform Builder Help. No additional custom API information is required for the features currently supported in the Direct3D Mobile driver.

Reference information on basic Direct3D Mobile driver functions, methods, and structures can be found at the following location in the PB Help documentation:

Developing a Device Driver –> Windows CE Drivers –> Direct3D Mobile Display Drivers –> Direct3D Mobile Driver Reference

Reference information on Direct3D Mobile functions, callbacks, and structures can be found at the following location in the PB Help documentation:

Windows CE Features –> Graphics and Multimedia Technologies –> Graphics –> Direct3D Mobile.

18.7.2 OpenGL ES

Documentation for the OpenGL driver APIs can be found at http://www.khronos.org/opengles.

Chapter 19
NAND Flash Media Driver (FMD)

Windows CE provides driver support for flash media devices using FMD (Flash Media Driver) and FAL (Flash Abstraction Layer) software architecture. The FMD and FAL allow NAND flash storage to be exposed as a block driver that is accessed using file system. The FMD software layers ported to the MX31/MX32 NAND flash controller is responsible for the actual I/O with the corresponding NAND flash devices respectively. The NAND FMD driver included in the MX31/MX32 BSP is targeted for the NAND flash device shipped with the MX31/MX32 ADS kit. These drivers can be easily ported to other NAND flash devices.

The NAND flash device can be sorted as two categories, that are small page size(page size is 512 bytes) NAND device and large page size(page size is 2048 bytes) NAND device. For MX31 BSP, the large page size NAND (K9F1G08U0A) and small page size NAND (K9F1G08U0B) can be supported. For MX32 BSP, the large page size NAND (K9F1G08U0 and K9LAG08U0M) and small page size NAND (K9F1G08U0B) can be supported.

19.1 NAND FMD Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT) on CE 5	MX31, MX32ADS
Target Platform (TGTPLAT) on CE 6	iMX31ADS, iMX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path on CE 5 Platform Driver Path on CE 6	\WINCE500\PLATFORM\<*TGTPLAT*>\SRC\COMMON\NANDFMD \WINCE500\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\BLOCK\NANDFMD \WINCE600\PLATFORM\<*TGTPLAT*>\SRC\COMMON\NANDFMD \WINCE600\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\BLOCK\NANDFMD
Import Library	N/A
Driver DLL	nandfmd.dll

Catalog Item	For MX31 on CE 5: Third Party –> BSPs –> Freescale MX31:ARMV4I–> Storage Drivers –> MSFlash Drivers –> Samsung K9K1G08U0B NAND Flash. Third Party –> BSPs –> Freescale MX31:ARMV4I–> Storage Drivers –> MSFlash Drivers –> Samsung K9F1G08U0A NAND Flash. For MX32 on CE 5: Third Party–> BSPs –> Freescale MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9F1G08U0A NAND Flash. Third Party–> BSPs –> Freescale MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9K1G08U0B NAND Flash. Third Party–> BSPs –> Freescale MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9LAG08U0M NAND Flash. For MX31 on CE 6 Third Party –> BSP –> Freescale i.MX31 ADS:ARMV4I–> Storage Drivers –> MSFlash Drivers –> Samsung K9K1G08U0B NAND Flash. Third Party –> BSPs –> Freescale i.MX31 ADS:ARMV4I–> Storage Drivers –> MSFlash Drivers –> Samsung K9F1G08U0A NAND Flash. For MX32 on CE 6: Third Party–> BSP –> Freescale i.MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9F1G08U0A NAND Flash driver support Third Party–> BSP –> Freescale i.MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9K1G08U0B NAND Flash driver support Third Party–> BSP –> Freescale i.MX32 ADS:ARMV4I –> Storage Drivers –> MSFlash Drivers –> Samsung K9LAG08U0M NAND Flash driver support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NAND_FMD=1 BSP_NAND_K9K1G08U0B=1 or BSP_NAND_K9F1G08U0A=1 (for MX31) BSP_NAND_K9K1G08U0B=1 or BSP_NAND_K9F1G08U0A=1 or BSP_NAND_K9LAG08U0M=1 (for MX32)

19.2 Requirements

The NAND FMD should meet the following requirements:

1. Support the Windows CE FMD interface.
2. Support the NAND flash device included in the MX31/MX32 ADS kit.
3. Support EMI clock gating for power management.

19.3 Hardware Operation

Refer to the NAND Flash Controller (NFC) chapter in the specification and data sheet for the NAND flash device included with the hardware kit for detailed operation and programming information.

19.3.1 MX31 ADS Configuration

The MX31 ADS kit includes a NAND flash memory card that attaches to the MX31 CPU board. Populate the memory card on the MX31 CPU board to support the NAND FMD.

19.3.2 MX32 ADS Configuration

The MX32 ADS kit includes a NAND flash memory card that attaches to the MX32 CPU board. Populate the memory card on the MX32 CPU board to support the NAND FMD.

19.3.3 Conflicts with other SoC peripherals

19.3.3.1 MX31/MX32 Peripheral Conflicts

The NAND flash controller interface consists of shared EMI signals and NAND-specific signals. The NAND-specific signals (NFWE_B, NFRE_B, NFALE, NFCLE, NFWP_B, NFCE_B, NFRB) can be configured for alternate functionality (ATA, USB H2, GPIO) using the MX31/MX32 IOMUX. The configuration supported by the BSP does not use this alternate functionality and dedicates these signals for NAND flash controller use. Changing this configuration would result in a conflict and prevent proper operation of the NAND FMD.

19.4 Software Operation

The development concepts for flash media drivers are described in the **Platform Builder for Microsoft Windows CE 5.0/6.0 Help** under the topic *Developing a Device Driver > Windows CE Drivers > Flash Media Drivers*. The NAND FMD supported in the MX31/MX32 ADS BSPs implements the required FMD functions for interfacing to NAND flash devices.

19.4.1 Compile-Time Configuration Options

The NAND FMD driver abstacts the details of the NAND flash memory device to a single header file. This header file is found in the \WINCE500\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMD or \WINCE600\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMDdirectory and named according to the NAND device. The configuration required for the NAND devices that ship with the ADS hardware kits has already been provided in the BSP.

To support a different NAND device, create a new header using one of the existing NAND device headers as a template and update the device-specific information. Then update the reference to the device-specific header in \WINCE500\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMD\nandfmd.h or \WINCE600\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMD\nandfmd.h and recompile the NAND FMD driver for the new device.

19.4.2 Loading and Initialization

The loading and initialization of the NAND FMD is primarily controlled using registry keys. Refer to the **Platform Builder for Microsoft Windows CE 5.0/6.0 Help** under the topic *Windows CE Features > File Systems and Data Store > File Systems and Data Store Registry Settings*.

19.4.3 Registry Settings

The registry keys implemented for the NAND FMD provides basic support for loading and configuring the NAND as a file system mount. Many more configuration options are available and are discussed in the **Platform Builder for Microsoft Windows CE 5.0/6.0 Help** under the topic *Windows CE Features -> File Systems and Data Store -> File Systems and Data Store Registry Settings*.

19.4.4 DMA Support

The NAND FMD currently does not provide DMA support. The conditional compilation variable BSP_SDMA_SUPPORT_NANDFC found in the BSP header bsp_cfg.h for the MX31ADS, MX32ADS are ignored.

19.4.5 Power Management

The power management support provided by the NAND FMD leverages clock gating features available within the hardware. The NAND flash controller is a component of the EMI (External Memory Interface). The clock gating for the EMI is managed globally for all EMI components. This implies that while the NAND flash controller does not have individual clock gating capability, the clocks to the NAND flash controller will be disabled when the EMI is clock gated.

The CSPDDK manages the configuration of the EMI clock gating. The NAND FMD driver is responsible for informing the CSPDDK when the NAND flash interface is active. This is achieved by using the DDKClockSetGatingMode within the FMD functions to signal when EMI clocks must remain active for accessing the NAND memory device.

19.5 Driver Testing

The NAND FMD was tested using Windows CE 5.0/6.0 Test Kit and additional system use cases. This section will describe the test scenarios that were used to verify the operation of the NAND FMD.

19.5.1 Test Hardware

The following table lists the required hardware to run the driver tests.

Requirements	Description
8-Bit NAND FLASH Card	Samsung K9K1G08U0B memory card included in MX31 ADS kit for testing with MX31 BSP. Samsung K9F1G08U0A memory card included in MX31 ADS kit for testing with MX31 BSP. Samsung K9K1G08U0B memory card included in MX32 ADS kit for testing with MX32 BSP. Samsung K9F1G08U0A memory card included in MX32 ADS kit for testing with MX32 BSP. Samsung K9LAG08U0M memory card included in MX32 ADS kit for testing with MX32 BSP.

19.5.2 CETK Testing

The CETK includes Storage Device tests that can be used to exercise the NAND FMD. The following table lists the CETK tests that were performed and provides the test configuration necessary to target the NAND FMD.

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.

CETK Test	Command Line
Other Tests > Partition Driver Test	tux -o -d msparttest -c “-z”
Storage Device > Storage Device Block Diver Read/Write Test	tux -o -d rwtest -c “-z” Note: This test does not recognize the storage device profile parameter. You may need to remove other storage devices from the image so that the device targeted for the test appears as DSK1 on the system.
Storage Device > Storage Device Block Diver Benchmark Test	tux -o -d rw_all -c “-p FlashDisk -z”
Storage Device > Storage Device Block Diver API Test	tux -o -d disktest -c “-p FlashDisk -z”
Storage Device > Flash Memory Read/Write and Performance Test	For WinCE CETK: tux -o -d flshwear -c “-z” For Mobile CETK: tux -o -d flshwear -c “-z /profile FlashDisk”
Storage Device > File System Driver Test	tux -o -d fsdtst -c “-p FlashDisk -z”

19.5.3 System Testing

The following system tests were performed to verify the operation of the NAND FMD:

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/from the NAND storage.

Write media files to NAND storage. Use Windows Media Player to playback media files from NAND storage.

Chapter 20
Notification LED Driver

The Notification LED is mainly used to notify the user about the occurrence of an event.

20.1 Notification LED Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31,MX32
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\NLEDDRVR
Import Library	N/A
Driver DLL	nleddrvr.dll
Catalog Item	N/A
SYSGEN Dependency	SYSGEN_NLED=1
BSP Environment Variables	Remove BSP_NONLED

20.2 Requirements

The Notification LED driver should meet the following requirements:

- The driver shall support only one Notification LED (STAT0) for MX31 and MX32ADS.

20.3 Hardware Operation

Refer to the Peripheral Bus Controller document for hardware implementation details for the Notification LED.

20.3.1 Conflicts with other SoC peripherals

20.3.1.1 MX31, MX32 Peripheral Conflicts

No conflicts.

20.4 Software Operation

20.4.1 Communicating with the Notification LED

The Notification LED is a stream interface driver, and is thus accessed through the file system APIs. To communicate with the NLED, 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.

20.4.2 Creating a Handle to the Notification LED

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

To open a handle to the NLED:

1. Insert a colon after the NLED for the first parameter, *lpFileName*.
— For example, specify NLD1: for STAT0.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an NLED 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.

The following code example shows how to open an NLED.

```
// Open the NLED.
HANDLE hNLeddrvr;
hNLeddrvr = CreateFile (TEXT ("NLD1:"),                    // name of device
                        GENERIC_READ|GENERIC_WRITE,        // desired access
                        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)
```

20.4.3 Configuring the Notification LED

Once the handle for the driver is obtained, it can be used for configuring parameters used by the NLED driver by calling the **DeviceIoControl** function with appropriate IOCTL.

The configuration parameters are defined in the structure **NLED_SETTINGS_INFO**. It is possible to set the time cycle of a blink, ON time of the cycle, OFF time of the cycle, number of ON blink cycles & number of OFF blink cycles. Please refer to Platform Builder documentation for more details about the structure: **Developing a Device Driver –> Windows CE Drivers –> Notification LED Drivers –> Notification LED Driver Reference –> Notification LED Driver Structures**

Please refer to section **NLED Driver API Reference** for more details on the NLED Driver IOCTLs.

20.4.4 Closing the Handle of the Notification LED

Call the CloseHandle( ) function to close a handle to the NLED when an application is done using it. CloseHandle has one parameter, the handle returned by the CreateFile function that opened the NLED.

```
CloseHandle (hNLeddrvr);
```

20.4.5 Power Management

There is no clock that needs to be enabled for this module.

20.4.5.1 PowerUp

This function will bring back the state of the Notification LED to its original state i.e. to a state just before the power down sequence happened.

20.4.5.2 PowerDown

This function will turn off the NLED to save the power and is called while entering the suspend state.

20.4.5.3 IOCTL_POWER_CAPABILITIES

N/A

20.4.5.4 IOCTL_POWER_SET

N/A

20.4.5.5 IOCTL_POWER_GET

N/A

20.4.6 Notification LED Registry Settings

The following registry keys are required to properly load the Notification LED driver.

```
; These registry entries load the NLed driver. The IClass value must match
; the NLED_DRIVER_CLASS definition in nled.h -- this is how the system
; knows which device is the Notification LED driver.

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\NLed]
   "Prefix"="NLD"
   "Dll"="nleddrvr.dll"
   "Flags"=dword:8                     ; DEVFLAGS_NAKEDENTRIES
   "Order"=dword:0
   "Index"=dword:1
   "IClass"="{CBB4F234-F35F-485b-A490-ADC7804A4EF3}"

[HKEY_LOCAL_MACHINE\System\Events]
    "SYSTEM/NLedAPIsReady"="Notification LED APIs"
```

20.5 Unit Test

The NLED CETK test cases verify the functionality of NLED driver.

20.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
No additional hardware required	

20.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
Nleddrvrtest.dll	Test .dll file

20.5.3 Building the NLED Tests

In order to build the NLED tests, complete the following steps:

Build an OS image for the desired configuration:

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the NLED Tests directory. (\WINCE500\SUPPORT\TESTS\NLEDDRVR)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command at the prompt and press return.

After the build completes, the nleddrvrtest.dll file will be located in the $(_FLATRELEASEDIR) directory.

20.5.4 Running the NLED Tests

The command line for running the NLED tests is ***tux –o –d nleddrvrtest***. You can provide an additional option –f if you wish to redirect the test results to a file. The NLED tests do not contain any test specific command line options.

The following table describes the test cases contained in the NLED tests.

Test Case	Description
NLED On Test	Illuminates the NLED.
NLED Blink Test	Tests that the NLED blinks for the specified turn On and turn Off time.
NLED Off Test	Turns the NLED off the NLED.

20.6 NLED Driver API Reference

20.6.1 NLED Driver IOCTLS

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

20.6.1.1 IOCTL_NLED_GETDEVICEINFO

This **DeviceIoControl** request can retrieve NLED device information. This information includes the number of NLEDs supported by the driver, NLED support information and NLED settings information.

The kind of information required needs to be specified in the *lpInBuffer* parameter of the **DeviceIoControl** request.

Parameters

lpInBuffer

Pointer to a buffer that contains the information request. This request can be anyone of the following:

NLED_COUNT_INFO_ID

NLED_SUPPORTS_INFO_ID

NLED_SETTINGS_INFO_ID

Please refer to Platform Builder documentation for more details on using these NLED structures: **Developing a Device Driver −> Windows CE Drivers −> Notification LED Drivers −> Notification LED Driver Reference**

nInBufferSize

Buffer to have a size of UINT.

lpOutBuffer

Pointer to NLED_SUPPORTS_INFO structure where the queried information would be stored.

nOutBufferSize

Size in bytes of the structure NLED_SUPPORTS_INFO.

20.6.1.2 IOCTL_NLED_SETDEVICE

This **DeviceIoControl** is used to change the configuration settings of the LED. The data that needs to be updated is stored in the input buffer which is pointed to by the *lpInBuffer* parameter of the **DeviceIoControl** request.

Parameters

lpInBuffer

Pointer to a buffer that contains the data that is to be updated. This data is the NLED_SETTINGS_INFO structure which contains the new configuration detatils. Refer to Platform Buider documentation for detailed information about this structure: **Developing a Device Driver −> Windows CE Drivers −> Notification LED Drivers −> Notification LED Driver Reference −> Notification LED Driver Structures**

nInBufferSize

Size in bytes of the structure NLED_SETTINGS_INFO.

Chapter 21
One-Wire Interface (OWIRE)

The One-wire Interface provides a communication line through which it is possible for the ARM core to communicate with the DS2433 4-Kbit 1-Wire EEPROM by writing or reading one bit data at a time. This DS2433 EEPROM holds the battery characteristics information.

21.1 One-Wire Interface Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\OWIRE
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\OWIRE
Import Library	N/A
Driver DLL	owire.dll
Catalog Item	Third Party —> BSPs —> Freescale <*TGTPLAT*>—> Device Drivers —>One-wire
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_OWIRE = 1

21.2 Requirements

The One-wire driver should meet the following requirements:

1. The DS2433 EEPROM uses its own protocol for communicating with One-wire Interface and the driver shall support this protocol.
2. The driver shall support two power management modes, full on and full off.

21.3 Hardware Operation

Refer to the chapter on One-wire Interface in the hardware specification document for detailed operation and programming information.

21.3.1 Conflicts with other SoC peripherals

21.3.1.1 MX31 and MX32 Interface Conflicts

One-Wire interface uses the BATT_LINE pad, which does not have any conflict.

In order to use One-wire Interface to communicate with the DS2433 4-Kbit EEPROM, jumper JP14 must be connected properly to connect pins 1 & 2. This configuration connects the DS2433 EEPROM to the One-wire Interface data pin.

21.4 Software Operation

21.4.1 Communicating with the One-Wire Interface

The One-wire Interface module is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the One-wire interface, 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 basic steps are detailed below.

21.4.2 Creating a Handle to the One-Wire Interface

Call the **OwireOpenHandle( )** function that is defined in the driver file. This function will in turn call the **CreateFile** function to open a connection to the One-wire Interface. If a One-wire port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the One-wire Interface:

Call the **OwireOpenHandle( )** function which would return a handle to the One-wire Interface.

The following code example shows how to open a One-wire device.

```
// Open handle to the OWIRE device
 hOwire = OwireOpenHandle ();
 if (hOwire == NULL)
 {
     g_pKato->Log(1, (TEXT("InitializeTests: OwireOpenHandle failed!\r\n")));
     return TPR_FAIL;
 }
```

21.4.3 Configuring the One-Wire Interface

N/A

21.4.4 Write Operations

Before initiating the write operation on the One-wire Interface bus, it is required to reset the One-wire device and wait for the Device Presence Pulse to ensure that DS2433 EEPROM is connected to the interface. This can be accomplished by calling the driver function **OwireResetPresencePulse( ),** along

with a handle to the One-wire Interface passed as a parameter. This **OwireResetPresencePulse( )** will in turn issue an OWIRE_IOCTL_RESET_PRESENCE_PULSE IOCTL which would be handled by the driver.

The following code example shows this presence detect operation.

```
if (!OwireResetPresencePulse(hOwire))
  {
      g_pKato->Log(OWIRE_ZONE_ERROR,
          (TEXT("OwireWriteMemTest: OwireResetPresencePulse() failed!\r\n")));
      return TPR_FAIL;
  }
```

Once a DS2433 EEPROM is detected, we can continue with the write operation. The One-wire Interface write operation involves exchange of commands and responses, before the data is finally written into the EEPROM. **OwireWrite()** is used to send the protocol command or data to the the DS2433 EEPROM. **OwireRead()** is used to receive the response or data from the DS2433 EEPROM. The following steps are used to write data into DS2433 EEPROM memory:

- Writing the data to DS2433 EEPROM scratchpad (send WRITE SCRATCHPAD command)
- Issue reset on the bus and wait for presense pulse (indicates to DS2433 EEPROM that there are no more bytes to write)
- Reading back the data from the DS2433 EEPROM scratchpad and verifying (send READ SCRATCHPAD command, then receive the response)
- Confirming the DS2433 EEPROM to copy the data from scratchpad to the memory (send COPY SCRATCHPAD command)
- Wait for 5 ms
- Optionally, reading the data from memory and verifying (send READ MEMORY command, then receive the memory contents)

Please refer to the DS2433 EEPROM reference manual for more information on the protocol commands that needs to be issued before performing any operation on DS2433 EEPROM.

To write command / data into DS2433 EEPROM using One-wire Interface:

1. Create a write buffer of bytes to hold the DS2433 EEPROM commands and the actual data.
2. Call the driver function **OwireWrite( )** along with the following set of parameters:

- Handle to the One-wire interface which was previously acquired using the **OwireOpenHandle( )** function.
- Address of the write buffer that was created.
- The number of bytes that were written into the write buffer.

This function will in turn call the **WriteFile** API to write the command and the data into the One-wire Interface bus.

The following code example shows WRITE SCRATCHPAD command being issued.

```
BYTE writeBuffer[8] = {0xCC, 0x0F, 0x40, 0x00, 0xA0, 0xA1, 0xA2, 0xA3};
                          //0xCC->SKIP ROM command; 0x0F->WRITE SCRATCHPAD command
                          //0x40 & 0x00->Starting Address (0X0040); 0xA0 0xA1 0xA2 0xA3-> data
```

```
if (!OwireWrite(hOwire, writeBuffer, 8))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireWriteMemTest: OwireWrite() failed!\r\n")));
        return TPR_FAIL;
    }
```

The following code example shows READ SCRATCHPAD command being issued.

```
BYTE writeBuffer[2] = {0xCC, 0xAA};
                          //0xCC->SKIP ROM command; 0xAA->READ SCRATCHPAD command

if (!OwireWrite(hOwire, writeBuffer, 2))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireWriteMemTest: OwireWrite() failed!\r\n")));
        return TPR_FAIL;
    }
```

The following code example shows receiving the response of READ SCRATCHPAD command.

```
BYTE readBuffer[7];

if (!OwireRead(hOwire, readBuffer, 7))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireWriteMemTest: OwireRead() failed!\r\n")));
        return TPR_FAIL;
    }
```

In case of successful write to the scratchpad, the response should contain these 7 bytes: 0x40 0x00 (address), 0x03 (value of E/S register in DS2433 EEPROM after successfully writing the 4 bytes from offset 0 of the scratchpad), 0xA0 0xA1 0xA2 0xA3 (the data written into the scratchpad in the previous step).

The following code example shows COPY SCRATCHPAD command being issued.

```
BYTE writeBuffer[5] = {0xCC, 0x55, 0x40, 0x00, 0x03};
                          //0xCC->SKIP ROM command; 0x55->COPY SCRATCHPAD command
                          //0x40, 0x00, 0x03: first 3 bytes received from READ SCRATCHPAD,
                          //which is used by DS2433 as authorization code for write to EEPROM

if (!OwireWrite(hOwire, writeBuffer, 5))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireWriteMemTest: OwireWrite() failed!\r\n")));
        return TPR_FAIL;
    }
```

21.4.5 Read Operations

Before initiating the read operation on the One-wire Interface bus, it is required to reset the One-wire interface and wait for the Device Presence Pulse to ensure that DS2433 EEPROM is connected to the interface. This can be accomplished by calling the driver function **OwireResetPresencePulse( ),** along with a parameter which is a handle to the One-wire interface. This **OwireResetPresencePulse( )** will in turn issue an IOCTL OWIRE_IOCTL_RESET_PRESENCE_PULSE which would be handled by the driver.

The following code example shows this presence detect operation.

```
if (!OwireResetPresencePulse(hOwire))
  {
      g_pKato->Log(OWIRE_ZONE_ERROR,
          (TEXT("OwireWriteMemTest: OwireResetPresencePulse() failed!\r\n")));
      return TPR_FAIL;
  }
```

Once the DS2433 EEPROM connection is detected, we can continue with the read operation. The One-wire Interface write operation involves exchange of commands and responses, before the data is finally read from the EEPROM. **OwireWrite()** is used to send the protocol command or data to the the DS2433 EEPROM. **OwireRead()** is used to read the response or data from the DS2433 EEPROM. The following steps are used to read data from DS2433 EEPROM memory:

- Read the memory (send READ MEMORY command, then receive the memory contents)

Please refer to the DS2433 EEPROM reference manual for more information on the protocol commands that needs to be issued before performing any operation on DS2433 EEPROM.

The following code example shows the READ MEMORY command being issued.

```
BYTE writeBuffer[4] = {0xCC, 0xF0, 0x40, 0x00} ; //0xCC->SKIP ROM command; 0xF0->READ MEMORY
                                    //command; 0x40 & 0x00->Starting Address (0X0040);
    if (!OwireWrite(hOwire, writeBuffer, 4))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireReadMemTest: OwireWrite failed!\r\n")));
        return TPR_FAIL;
    }
```

Once this READ command is issued, the bus master will start issuing read time slots and receive data from DS2433 EEPROM starting from the address 0x4000.

To read data from DS2433 EEPROM using One-wire Interface:

1. Create a read buffer of bytes to hold the data read from DS2433 EEPROM.
2. Call the driver function **OwireRead( )** along with the following set of parameters:

- Handle to the One-wire Interface which was previously acquired using the **OwireOpenHandle( )** function.
- Address of the read buffer that was created.
- The number of bytes that is to be read from the read buffer.

This function will in turn call the **ReadFile( )** API to read the data from the DS2433 EEPROM.

After issuing the READ MEMORY command as described previously, following code can be used to read 8 bytes of data from address 0x0040 of DS2433 EEPROM.

```
if (!OwireRead(hOwire, readBuffer, 8))
    {
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireReadMemTest: OwireRead failed!\r\n")));
        return TPR_FAIL;
    }
```

21.4.6 Closing the Handle to the One-Wire Interface

Call the **OwireCloseHandle( )** driver function to close a handle to the One-wire Interface when an application is done using it. This function will in turn call the **CloseHandle** API.

OwireCloseHandle( ) has one parameter, which is the handle returned by the **OwireOpenHandle( )** function call that opened the One-wire Interface.

21.4.7 Power Management

The primary method for limiting power consumption in the One-wire module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. One-wire module clock is enabled only when an instance of the module is created in the WIR_Open( ) and disabled in the WIR_Close( ) function when the module instance is closed.

Moreover, the One-wire module goes into low power mode by gating off the clock automatically whenever it is not in use i.e. whenever it is not communicating with DS2433 EEPROM. Refer to the chapter on One-wire Interface in the hardware specification document for detailed description of this.

21.4.7.1 PowerUp

This function is not implemented for the One-wire driver.

21.4.7.2 PowerDown

This function is not implemented for the One-wire driver.

21.4.7.3 IOCTL_POWER_CAPABILITIES

N/A

21.4.7.4 IOCTL_POWER_SET

N/A

21.4.7.5 IOCTL_POWER_GET

N/A

21.4.8 One-Wire Interface Registry Settings

The following registry setting is required to be added in the platform.reg file to properly load the One-wire interface module.

```
IF BSP_OWIRE
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\OWIRE]
   "Prefix"="WIR"
   "Dll"="owire.dll"
   "Index"=dword:1
```

```
ENDIF
```

21.5 Unit Test

The One-wire tests verify that the One-wire driver properly initializes the One-wire support.

21.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
The <TGTPLAT> board with DS2433 EEPROM as One-wire chip.	

21.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
OnewireTest.dll	Test .dll file

21.5.3 Building the One-Wire Tests

In order to build the One-wire tests, complete the following steps:

1. Build an OS image for the desired configuration
2. Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
3. Change to the OWIRETest directory. (\WINCE500\SUPPORT\TESTS\OWIRETest).
4. Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
5. Enter the build command at the prompt and press return.

After the build completes, the OnewireTest.dll file will be located in the $(_FLATRELEASEDIR) directory.

21.5.4 Running the One-Wire Tests

The command line for running the One-wire tests is ***tux –o –d OnewireTest***. The One-wire tests do not contain any test specific command line options.

The following table describes the test cases contained in the One-wire tests.

Test Case	Description	Parameters	Remarks
OwireReadStatus Test	Reads the DS2433 EEPROM's family code, unique 48-bit serial number and 8-bit CRC.	None	None
Owire Write Memory	Writes data into the DS2433 EEPROM & reads back the data and verifies whether the data was written correctly into the memory.	None	This test will skip if the <TGTPLAT> board uses DS2502 instead of DS2433 EEPROM, because DS2502 needs 12V to perform write, which is not supported by <TGTPLAT>.
Owire Read Memory	Reads data from DS2433 EEPROM's entire 4Kbit (512 bytes) memory and displays.	None	This test will skip if the <TGTPLAT> board uses DS2502 instead of DS2433 EEPROM, because DS2502 needs 12V to perform write, which is not supported by <TGTPLAT>.
Owire Soft Reset	This test issues OWIRE_IOCTL_RESET IOCTL and test for the OWIRE module soft reset functionality.	None	None

21.6 One-Wire Driver API Reference

21.6.1 One-Wire Driver IOCTLS

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

21.6.1.1 OWIRE_IOCTL_RESET

This **DeviceIoControl** is used to perform a software reset of the One-wire module.

Parameters

lpInBuffer NULL

nOutBufferSize 0

lpOutBuffer NULL

nOutBufferSize 0

21.6.1.2 OWIRE_IOCTL_RESET_PRESENCE_PULSE

This **DeviceIoControl** request issues a reset pulse and waits for as long as 1ms till it detects a presence pulse. If no presence signal is detected within 1ms, the function fails.

Parameters

lpInBuffer	NULL
nOutBufferSize	0
lpOutBuffer	NULL
nOutBufferSize	0

21.6.2 One-Wire Driver Structures

N/A

Chapter 22
PCMCIA Driver

The Windows CE 5.0 PC Card stack replaces the PCMCIA stack included in Windows CE .NET 4.2 and earlier. This stack exposes the bus agnostic drivers interface for clients as well as a legacy compatibility layer that supports all existing PCMCIA clients. The Windows CE 5.0 stack fully supports 16-bit PCMCIA cards and 32-bit Cardbus cards. The overall high level diagram is shown as follows:

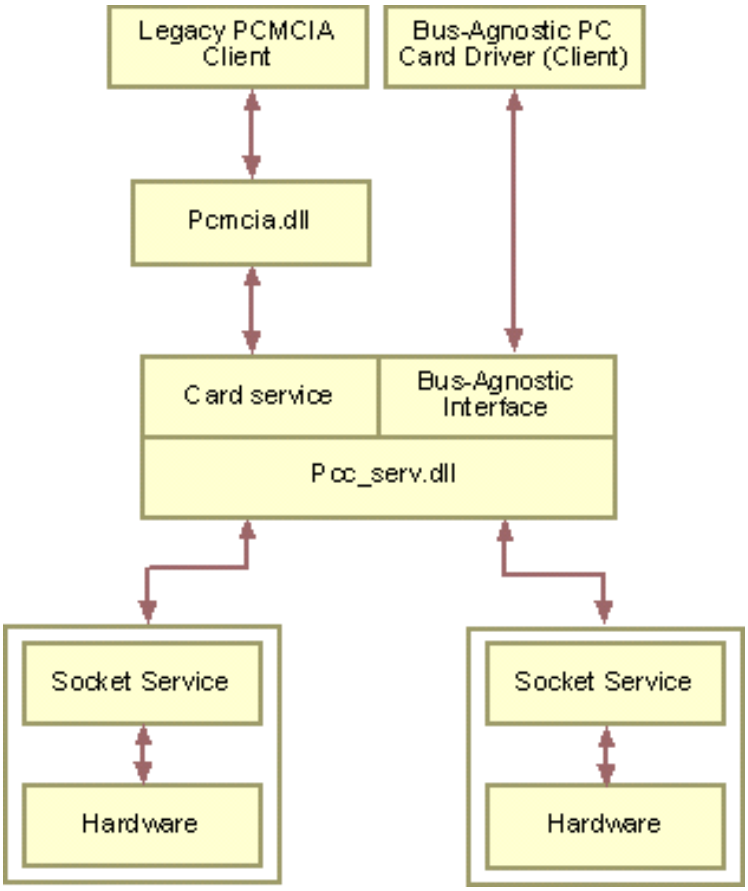


Figure 22-1. Windows CE 5.0 PC Card Driver Architecture

For the detailed description, please refer to the Platform Builder for Microsoft Windows CE 5.0 Help. It can be seen that the PC card bus driver actually provides the card service and bus-agnostic interface. The socket service is separated from the PC card bus driver and linked with the hardware related potion to form the socket driver. The PCMCIA driver we are talking about is actually this socket driver, which is the responsibility of the OEM developer. For 16 Bit PCMCIA controller, the actual modules involved in the above diagram are:

PC card bus driver (Card Service driver): Pcc_serv.dll｜ pcc_serv16.dll

PCMCIA Legacy driver wrapper: Pcmcia.dll｜ pcc_pcm.dll

socket driver: Pcc_<*TGTSOC*>.dll

The following sections will describe the detailed implementation of the socket driver (pcc_mx31).

22.1 PCMCIA Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31
Target SOC (TGTSOC)	MX31
MXARM11 CSP Driver Path	N/A
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\PCCARD
CSP Static Library	<*TGTSOC*>_pcc_lib.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\PCCARD
Import Library	pcc_com.lib, cspddk.lib, coredll.lib, ceddk.lib
Driver DLL	pcc_<*TGTSOC*>.dll
Catalog Item	Third Party －> BSPs －> Freescale <*TGTPLAT*> －> Device Drivers －> PCCard －> PC Card Host －><*TGTPLAT*> PCMCIA
SYSGEN Dependency	SYSGEN_ATADISK=1(For Compact Flash/PC Card Storage Devices)
BSP Environment Variables	BSP_PCMCIA=1, BSP_SERIAL_UART5=0, BSP_USB_HSH2=0, BSP_SDCH2=0

22.2 Requirements

The PCMCIA driver should meet the following requirements:

1. The driver shall conform to the PC Card Driver Architecture in Windows CE 5.0.
2. The driver shall support 16 bit PCMCIA cards only.
3. The driver currently supports 4 windows out of the maximum 5. Out of the 4 windows, 2 windows are configured as Memory windows and the other 2 as IO windows.
4. Since card status change and client interrupts share the same IRQ and Windows CE 5.0 PCMCIA client drivers need full control of the client interrupt (creating interrupt event, calling InterruptInitialize() and creating the IST thread), the current socket driver has to use the polling mode for checking the card status changes. PCMCIA client drivers use interrupt mode by default though polling can be used also.
5. The driver shall support two power management modes, full on and full off.

22.3 Hardware Operation

Refer to the chapter on the PCMCIA (Personal Computer Memory Card International Association) in the hardware specification document for detailed operation and programming information. Also few of PCMCIA pins are connected to the processor through PBC (Peripheral Bus Controller). Refer to the Peripheral Bus Controller CPLD document for detailed operation and programming information.

22.3.1 Conflicts with other SoC peripherals

22.3.1.1 MX31 Peripheral Conflicts

PCMCIA has pin conflicts with the SDHC 2 controller, USB Host 2 controller, MSHC 2 controller and UART5 module. When the PCMCIA is configured in the BSP (selected from the catalog), none of these other devices should be configured.

Note: PCMCIA does not work on MX31 CPU boards with CPU Rev G and above (MX31 Silicon 1.2 and above). One of the PCMCIA signal was acting as DDR Control signal, so to have a risk free DDR, PCMCIA support has been removed from MX31 CPU Rev G onwards.

22.3.1.2 To run with KITL disabled/ from NOR flash on MX31

1)Make sure NAND support (IMGNAND environment variable) is removed from the image. Also make sure necessary PCMCIA client drivers are added to the image

2)Make sure switch SW3-8 on MX31 base board should be OFF to enable passive KITL mode.

3)Please refer “Downloading and Debugging Images”, Page 19 of WinCEBSPUserGuide.pdf for flashing NOR image using EBOOT.

22.4 Software Operation

The PCMCIA driver follows the Microsoft-recommended architecture for PC Card drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver –> Windows CE Drivers –> PC Card Drivers –> PC Card Development Concepts**.

The socket driver is the build-in device driver which exports the following functions from the DLL:

- Init()
- Deinit()
- PowerUp()
- PowerDown()

And the socket service APIs are encapsulated in SS_SOCKET_SERVICE structure which are passed to card service driver by using CS_AddSocket() for future accessing. The main API calls between card service driver and socket service driver are summarized as follows:

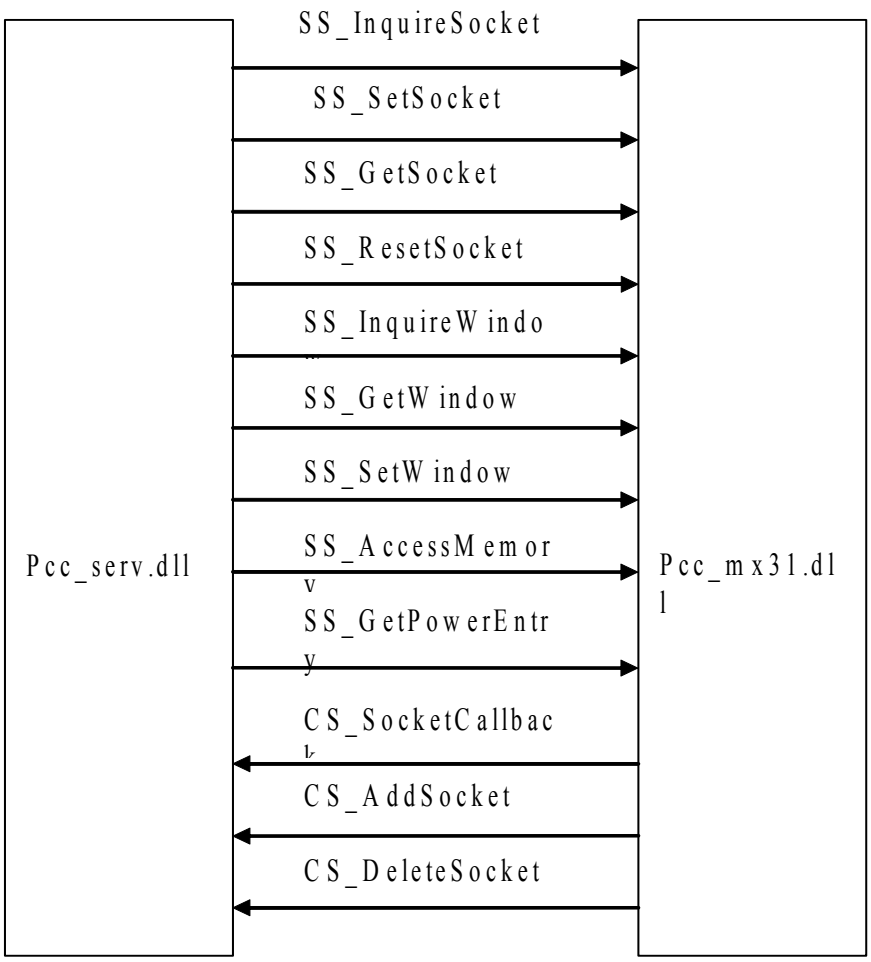


Figure 22-2. Communication between Card Service Driver and Socket Service Driver

The detailed description for the APIs of card service and socket service can be referenced from Platform Builder for Microsoft Windows CE 5.0 Help-PC Card Driver Reference.

The driver is designed with object oriented method.

22.4.1 Power Management

The primary method for limiting power consumption in the PCMCIA module is to gate off all clocks internal to PCMCIA. This brings the PCMCIA module to “listening” mode, where it waits for an indication that a card has been inserted. This is accomplished by writing 1 into **LPMEN** (Low Power Mode Enable) bit of **PGCR** (PCMCIA General Control register) register. In Low power mode, only card status change interrupts are generated (in order to wake up the core from stop on card detect, for example) and any card access in this mode will result in an error. When the PCMCIA module is reset, by default it stays in low power mode.

22.4.1.1 PowerUp

This function is implemented for the PCMCIA driver. The driver resets the PCMCIA module and generates a fake card insertion event to check if the card has already been inserted.

22.4.1.2 PowerDown

This function is implemented for the PCMCIA driver. The driver brings the PCMCIA module to low power mode by writing 1 into LPMEN bit of PGCR register.

22.4.1.3 IOCTL_POWER_SET

This function is not implemented for the PCMCIA driver, since the driver is not a stream interface driver.

22.4.2 PCMCIA Registry Settings

22.4.2.1 Platform specific registry setting to load PCMCIA driver

```
#if defined BSP_PCMCIA
#include "$(_TARGETPLATROOT)\src\drivers\pccard\pcc_mx31.reg"
#include "$(_PUBLICROOT)\common\oak\drivers\pccard\mdd\pcc_serv.reg"
[HKEY_LOCAL_MACHINE\Drivers\PCCARD\PCMCIA\TEMPLATE\PCMCIA]
   "Dll"="pcmcia.dll"
   "NoConfig"=dword:1
   ;"NoISR"=dword:1 ; Do not load any ISR.
   "IClass"=multi_sz:"{6BEAB08A-8914-42fd-B33F-61968B9AAB32}=PCMCIA Card Services"
#endif
```

22.4.2.2 PCMCIA specific registry settings (pcc_mx31.reg)

These registry provides configuration information to the driver about windows, polling or interrupt mode.

Since card status change and client interrupts share the same IRQ, Card Status changes are monitored through polling mode. For the future enhancement, it can be designed that card status change and client interrupts are using different SYSINTR so that the socket driver can use interrupt mode instead of polling mode.

ForceClientSysIntr and *ClientIRQ* are dependent on each other to enable interrupt mode for PC Card Client drivers.

CSCSysIntr, *PollingMode* and *PollTimeout* are dependent on each other to enable polling for PC Card status changes.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\pcc_mx31]
   "Dll"="pcc_mx31.dll"
   "Order"=dword:0
   "ForceClientSysIntr"=dword:1 ;Force to dynamic request the Client SysIntr
   "ClientIrq"=dword:36         ;Physical interrupt ID for client interrupt –IRQ_PCMCIA
;  "ClientSysIntr"=dword:26     ;Statically assigned Client System Interrupt ID
;  "CSCIrq"=dword:36            ;Physical interrupt ID for card status change interrupt
   "CSCSysIntr"=dword:26        ; Statically assigned System Interrupt ID
   "PollingMode"=dword:1
   "PollTimeout"=dword:1f4
```

```
   "PCCARDDLL"="pcc_serv.dll"

"IClass"=multi_sz:"{57430CF2-A260-4c9b-8F5C-FEF89217FE7C}=%b","{D0C3B9F2-D506-4c93-884D-6D71C
47AC9AA}=%b"

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\pcc_mx31\WindowEntry1]
    "RepeatCount"=dword:1                ; 1 instance of the memory window
    "WindowInfoWindowCaps"=dword:3       ; WIN_CAP_COMMON | WIN_CAP_ATTRIBUTE
    "WindowInfoMemoryCaps"=dword:19      ; MEM_CAP_8BIT | MEM_CAP_16BIT | MEM_CAP_PRG_BASE
    "WindowInfoMemMinSize"=dword:10      ; require 16 bytes alignment for TO3
    "WindowInfoMemMaxSize"=dword:200000  ; 2M
    "WindowInfoMemBase"=dword:BC200000   ; base address of the first memory window,
    ; the base addresses of other windows will be calculated if RepeatCount > 1
    "WindowInfoMemOffset"=dword:0
    "WindowInfoMemFirstByte"=dword:BC200000
    "WindowInfoMemLastByte"=dword:BC3FFFFF
    "WindowInfoMemGranularity"=dword:1
    "WindowInfoSlowest"=dword:97   ;WIN_SPEED_EXP_10MS|WIN_SPEED_MANT_12|WIN_SPEED_USE_WAIT
    "WindowInfoFastest"=dword:90   ;WIN_SPEED_EXP_100NS|WIN_SPEED_MANT_12|WIN_SPEED_USE_WAIT
    "WindowStateSpeed"=dword:88    ;WIN_SPEED_EXP_100NS|WIN_SPEED_MANT_10|WIN_SPEED_USE_WAIT
    "WindowStateState"=dword:2
    ;"WindowStateSize"=dword:200000
    "WindowStateBase"=dword:BC200000
    "WindowStateOffset"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\pcc_mx31\WindowEntry2]
    "RepeatCount"=dword:1                   ; 1 instance of the memory window
    "WindowInfoWindowCaps"=dword:3          ; WIN_CAP_COMMON | WIN_CAP_ATTRIBUTE
    "WindowInfoMemoryCaps"=dword:19         ; MEM_CAP_8BIT | MEM_CAP_16BIT | MEM_CAP_PRG_BASE
    "WindowInfoMemMinSize"=dword:10        ; require 16 bytes alignment for TO3
    "WindowInfoMemMaxSize"=dword:200000     ; 2M
    "WindowInfoMemBase"=dword:BC400000      ; base address of the first memory window,
    "WindowInfoMemOffset"=dword:0
    "WindowInfoMemFirstByte"=dword:BC400000
    "WindowInfoMemLastByte"=dword:BC5FFFFF
    "WindowInfoMemGranularity"=dword:1
   "WindowInfoSlowest"=dword:97  ;WIN_SPEED_EXP_10MS|WIN_SPEED_MANT_12|WIN_SPEED_USE_WAIT
    "WindowInfoFastest"=dword:90   ;WIN_SPEED_EXP_1NS|WIN_SPEED_MANT_12|WIN_SPEED_USE_WAIT
    "WindowStateSpeed"=dword:88   ;WIN_SPEED_EXP_1NS|WIN_SPEED_MANT_12|WIN_SPEED_USE_WAIT
    "WindowStateState"=dword:2
    ;"WindowStateSize"=dword:200000
    "WindowStateBase"=dword:BC400000
    "WindowStateOffset"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\pcc_mx31\WindowEntry3]
    "RepeatCount"=dword:1                   ; 1 instance of the IO window
    "WindowInfoWindowCaps"=dword:84         ; WIN_CAP_IO | WIN_CAP_WAIT
    "WindowInfoIOCaps"=dword:19             ; IO_CAP_8BIT | IO_CAP_16BIT | IO_CAP_PRG_BASE
    "WindowInfoIOMinSize"=dword:10          ; require 16 byte alignment for TO3
    "WindowInfoIOMaxSize"=dword:100000      ; 1M
    "WindowInfoIOFirstByte"=dword:BC000000  ; base address
    "WindowInfoIOLastByte"=dword:BC0FFFFF   ; base address + 1M - 1
    "WindowInfoIOGranularity"=dword:1

    "WindowStateState"=dword:5     ; WIN_STATE_MAPS_IO|WIN_STATE_16BIT
    "WindowStateSpeed"=dword:88    ; WIN_SPEED_EXP_1NS|WIN_SPEED_MANT_10|WIN_SPEED_USE_WAIT
    ;"WindowStateSize"=dword:100000
```

```
    "WindowStateBase"=dword:BC000000
    "WindowStateOffset"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\pcc_mx31\WindowEntry4]
    "RepeatCount"=dword:1                   ; 1 instance of the IO window
    "WindowInfoWindowCaps"=dword:84         ; WIN_CAP_IO | WIN_CAP_WAIT
    "WindowInfoIOCaps"=dword:19             ; IO_CAP_8BIT | IO_CAP_16BIT | IO_CAP_PRG_BASE
    "WindowInfoIOMinSize"=dword:10          ; require 16 byte alignment for TO3
    "WindowInfoIOMaxSize"=dword:100000      ; 1M
    "WindowInfoIOFirstByte"=dword:BC100000  ; base address
    "WindowInfoIOLastByte"=dword:BC1FFFFF   ; base address + 1M - 1
    "WindowInfoIOGranularity"=dword:1

    "WindowStateState"=dword:5       ; WIN_STATE_MAPS_IO|WIN_STATE_16BIT
    "WindowStateSpeed"=dword:88      ; WIN_SPEED_EXP_1NS|WIN_SPEED_MANT_10|WIN_SPEED_USE_WAIT
    ;"WindowStateSize"=dword:100000
    "WindowStateBase"=dword:BC100000
    "WindowStateOffset"=dword:0
```

22.5 Unit Test

The PCMCIA driver is tested using:

1. PC Card Legacy test cases. These test cases are included as part of the Windows CE 5.0 Test Kit (CETK). The test cases include PC Card 16-BIT PCMCIA Host Controller Driver Tests because the PCMCIA Module of i.MX31 only supports 16-BIT PCMCIA cards.
2. Storage Device test cases for PCMCIA storage card. These test cases are included as part of the Windows CE 5.0 Test Kit (CETK). The Storage Device test cases used to test PCMCIA driver include:

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

3. Ethernet test cases for PCMCIA Ethernet card. These test cases are included as part of the Windows CE 5.0 Test Kit (CETK). The Ethernet test cases used to test PCMCIA driver is the One-card Network Card Miniport Driver Test.
4. Wireless LAN test cases. These test cases are included as part of the Windows CE 5.0 Test Kit (CETK). The Wireless LAN test cases used to test PCMCIA driver is the One-card Network Card Miniport Driver Test.

NOTE

Storage Device Block Driver **Test case 4009** will fail. It is a known issue.
Please refer DDTS ticket DSPhl28459 for further details.

22.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
PCMCIA storage card	16 bit PC Card Flash storage card.
PCMCIA Ethernet card	16 bit PCMCIA Ethernet card (should be NE2000-compatible PCMCIA card)
PCMCIA Wireless LAN card	16 bit PCMCIA Wireless LAN card. (The supported WLAN drivers by WINCE are CISCO Aironet 340/350 or Intersil Prism2 or Realtek RTL8180 WLAN)

22.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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	Test .dll file used to perform File System Driver Test cases
disktest.dll	Test .dll file used to perform Storage Device Block Driver API test cases
rwtest.dll	Test .dll file used to perform Storage Device Block Driver Read/Write test cases
ddlx.dll	Test .dll file used to perform PC Card 16-BIT PCMCIA Host Controller Driver test cases
ndt_1c.dll, ndt.dll	Test .dll file used to perform the One-card Network Card Miniport Driver Test

22.5.3 Building the PCMCIA Tests

22.5.3.1 PC Card Legacy Test

The PC Card Tests come pre-built as part of the CETK. No steps are required to build these tests. The ddlx.dll file can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

22.5.3.2 Storage Device Test

The Storage Device Tests come pre-built as part of the CETK. No steps are required to build these tests. The fsdtst.dll, disktest.dll and rwtest.dll files can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

NOTE

The PC Card client driver for PCMCIA storage cards can be selected through the catalog item **Catalog –> Device Drivers –> Storage Devices –> Compact Flash / PC Card Storage (ATADISK).**

22.5.3.3 Ethernet Device Test

The Ethernet Device Tests come pre-built as part of the CETK. No steps are required to build these tests. The ndt.dll and ndt_1c.dll files can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

NOTE

The PC Card client driver for PCMCIA Ethernet cards can be selected through the catalog item **Catalog** –> **Device Drivers** –> **Networking** –> **Local Area Networking (LAN) devices** –> **NE2000-compatible (PCMCIA Card)**

22.5.3.4 Wireless LAN Device Test

The Wireless LAN Device Tests come pre-built as part of the CETK. No steps are required to build these tests. The ndt.dll and ndt_1c.dll files can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

NOTE

The PC Card client driver for PCMCIA WLAN cards can be selected through the catalog item **Catalog** –> **Device Drivers** –> **Networking** –> **Local Area Networking (LAN) devices.**

22.5.4 Running the PCMCIA Tests

22.5.4.1 PC Card Legacy Test

The command line for running the PC Card Test is:

tux –o –d ddlx -c"PCC16BitTest"

For detailed information on the Storage Device CETK test cases, see following sections of Platform Builder Help:

Debugging and Testing –> **Tools for Debugging and Testing** –> **Windows CE Test Kit** –> **CETK Tests** –> **Device Driver Loader and Tux Extender**

22.5.4.2 Storage Device Test

The command line for running the File System Driver Test is:

tux –o –d fsdtst –c “-p PCMCIA -zorch”

Note: a user can confirm the accessibility of this profile by running the File System Driver Test with the parameters ***–x1 –c “-z”***. The output will show a list of detected profiles – “PCMCIA” should be on the list.

The command line for running the Storage Device Block Driver API Test is:

tux –o –d disktest.dll –c -zorch -x4002-4008,4010-4018

The command line for running the Storage Device Block Driver Read/Write Test is:

tux –o –d rwtest.dll –c –zorch

For detailed information on the Storage Device CETK test cases, see following sections of Platform Builder Help:

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> File System Driver Test

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Storage Device Block Driver API Test

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Storage Device Block Driver Read/Write Test

22.5.4.3 Ethernet Device Test

The command line for running the One-card Network Card Miniport Driver test is:

tux –o –d ndt_1c.dll –c “-t NE20001”

For detailed information on the Storage Device CETK test cases, see following sections of Platform Builder Help:

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> One-card Network Card Miniport Driver Test

22.5.4.4 Wireless LAN Device Test

The command line for running the One-card Network Card Miniport Driver test is:

tux –o –d ndt_1c.dll –c “-t XXX” where XXX is the WLAN driver loaded.

For detailed information on the Storage Device CETK test cases, see following sections of Platform Builder Help:

Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> One-card Network Card Miniport Driver Test

22.6 PCMCIA Driver API Reference

The PCMCIA driver conforms to specification by Microsoft for the WINCE 5.0 PC Card drivers.

Chapter 23
Postfilter Driver

The Postfilter Driver provides an API to access to hardware acceleration for H.264 deblocking and MPEG4 deblocking and deringing. The Postfilter driver interfaces with the Image Processing Unit (IPU) Postfilter (PF) submodule. The Postfilter driver conforms to the architecture for Windows CE stream interface drivers.

23.1 Postfilter Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path on WINCE5.0 MXARM11 SOC Driver Path on WINCE6.0	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\IPU\PF ..\SOC\FREESCALE\MXARM11_FSL_V1\IPU\PF
CSP Driver Path	N/A
CSP Static Library on WINCE5.0 SOC Static Library on WINCE6.0	mxarm11_pf.lib pf_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\IPU\PF
Import Library	N/A
Driver DLL	pf.dll
Catalog Items	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PF=1 (default build)

23.2 Requirements

The Postfilter driver should meet the following requirements:

1. The driver shall support 3 postfiltering modes: H.264 deblocking, MPEG4 deblocking, and MPEG4 deblocking and deringing.

2. The driver shall support pause functionality for H.264 deblocking, allowing the programmer to specify a pause row on which the operation will be paused. The paused operation may then be resumed at any time.
3. The driver shall be a stream interface driver implementing the programming interface defined in this document
4. The driver shall support two power management modes, full on and full off.

23.3 Hardware Operation

Refer to the chapter on IPU in the hardware specification document for detailed operation and programming information.

23.3.1 Conflicts with other SoC peripherals

23.3.1.1 i.MX31/i.MX32 Peripheral Conflicts

There are no peripheral conflicts on this SoC.

23.4 Software Operation

23.4.1 Communicating with the Postfilter Driver

The Postfilter is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the Postfilter, a handle to the device must first be obtained using the **PFOpenHandle** function. Subsequent commands to the device are issued using various APIs supported by this driver.

23.4.2 Creating a Handle to the Postfilter Driver

To communicate with the PF driver, a handle to the device must first be created using the **PFOpenHandle** API. The default PF port is 1.

To open a Handle to the PF:

```
// Handle to the PF device
HANDLE g_hPF = NULL;

// opening the default PF port.
g_hPF = PFOpenHandle();
```

For more information on this API, please see the **PFOpenHandle** section under the PF API reference.

23.4.3 Configuring the Postfilter Driver

The **PFConfigure** API must be called to configure several important settings for the Postfilter driver. The **pfConfigData** data structure must be filled out and passed as a parameter to **PFConfigure**.

There are 3 important pieces of information needed to configure a Postfilter operation:

1. The postfiltering mode (e.g. H.264 deblocking or MPEG4 deblocking).
2. The input frame parameters, including width, height, and stride.
3. Parameters for the input buffer containing quantization parameter and boundary strength data, including the physical address and size of the buffer.

   Note: The Postfiltering hardware requires the physical address of a physically contiguous buffer. The **AllocPhysMem()** Windows CE API is recommended for allocating this physically contiguous buffer on Windows CE 5.0. We provide **PFAllocPhysMem()** API to allow a user mode Postfilter application to allocate a physically contiguous buffer on Windows CE 6.0.

The following code example shows how to configure the Postfilter driver.

```
// Configure Postfilter for MPEG4 deblocking
pfConfigData configData;
UINT32 iWidth, iHeight;

iWidth = 176;
iHeight = 144;
iQPBytesPerFrame = iHeight / 16 * iWidth / 16;

// Set up configuration data
configData.mode = pfMode_MPEG4Deblock;
configData.frameSize.width = iWidth;
configData.frameSize.height = iHeight;
configData.frameStride = iWidth;

configData.qpBuf = pQPPhysAddr; // Start address of a physically contiguous buffer
configData.qpSize = iQPBytesPerFrame;

PFConfigure(hPF, &configData);
```

23.4.4 Executing Postfilter Operations

Once the Postfilter driver has been configured, a postfiltering task can be commenced. A call to the **PFStart** function will begin the configured operation. **PFStart** takes as parameters information about the input and output buffers for the current postfiltering task. This information includes the size of the buffer, a pointer the physical address of the start of the Y data buffer, and offsets to the U and Y data buffers (**Note**: For the planar YUV data provided as input for postfiltering, the Y, U, and V data buffers must be physically contiguous in memory). Additionally, for the case of H.264 deblocking operations, a pause row may be specified. When the pause row is reached during the deblocking operation, the task will be paused, and will not resume until the **PFResume** API is called. To disable pausing, the pause row should be set to 0.

```
//-------------------------------------------------------------
// Set up Start Data Structure for MPEG4 deblocking operation
//-------------------------------------------------------------
pfStartParams startData;
```

```
pfBuffer inBuf, outBuf;
DWORD iYUVBytesPerFrame = iWidth * iHeight * 3/2;

// Set up input and output buffers.
inBuf.size = iYUVBytesPerFrame;
inBuf.yBufPtr = pInputFramePhysAddr; // Physical address of input buffer
inBuf.uOffset = iWidth * iHeight;
inBuf.vOffset = inBuf.uOffset * 5/4;

outBuf.size = iYUVBytesPerFrame;
outBuf.yBufPtr = pOutputFramePhysAddr; // Physical address of output buffer
outBuf.uOffset = inBuf.uOffset;
outBuf.vOffset = inBuf.vOffset;

startData.in = &inBuf;
startData.out = &outBuf;
startData.h264_pause_row = 0;

// Start PF
PFStart(hPF, &startData);
```

Three different events are signalled, representing the completion of 3 different phases of the Postfilter task: the completion of the Y component, the completion of the Cr componment, and the completion of the Cb component, which corresponds to the End-Of-Frame (EOF) of the Postfilter task. These events are signalled through named Windows CE Event objects. A WinCE Handle must be created, using either PF_Y_EVENT_NAME, PF_CR_EVENT_NAME or PF_EOF_EVENT_NAME strings defined in the pf.h header file, so that the application may be signalled when the postfilter task has completed. This handle will be used in a call to the **WaitForSingleObject** function.

The following sample code creates a handle to the Postfilter EOF event, and waits for that event to be signalled.

```
HANDLE g_hPFEOFEvent;

// Create event for Postfilter EOF
g_hPFEOFEvent = CreateEvent(NULL, FALSE, FALSE, PF_EOF_EVENT_NAME);

// Wait for End of Frame
WaitForSingleObject(g_hPFEOFEvent, INFINITE);
```

23.4.5 Closing the Handle to the Postfilter Driver

Call the **PFCloseHandle** function to close a handle to the Postfilter driver when an application is done using it.

23.4.6 Postfilter Registry Settings

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

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PF]
   "Prefix"="POF"
   "Dll"="pf.dll"
   "Order"=dword:20
   "Index"=dword:1
```

23.4.7 Power Management

The Postfilter driver consumes power primarily through the operation of the Postfilter IPU sub-module. If the Postfilter driver is included in the OS image, the Postfilter submodule will be enabled during boot-up, and will remain enabled until the system is shut down. No additional power management is supported at this time.

23.4.7.1 PowerUp

This function is not implemented for the Postfilter driver.

23.4.7.2 PowerDown

This function is not implemented for the Postfilter driver.

23.4.7.3 IOCTL_POWER_SET

This function is not implemented for the Postfilter driver.

23.5 Unit Test For MX31, MX32

The Postfilter driver unit tests verify the proper operation of the Postfilter driver modes of operation.

For H.264 deblocking mode, a full set of input and output reference data are provided to perform a bitmatch verification of the Postfilter operation. For MPEG4 operations, input data is provided, but no reference output data is provided. Thus, for MPEG4 postfiltering modes, an output YUV file is generated (and may be viewed using a YUV viewer too), but no bitmatching is performed.

23.5.1 Unit Test Software

The following table lists the required software to run the Postfilter driver tests on Windows CE 5.0.

Requirements	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
PFTest.dll	Main test .dll file.

The following table lists the required software to run the Postfilter driver tests on Windows CE 6.0.

Requirements	Description
pftest.exe	Postfilter test execution file.

23.5.2 Building the Postfilter Tests

23.5.2.1 Unit Test in Windows CE

To build the Postfilter unit test in Windows CE 5.0, complete the following steps.

Build an OS image for the desired configuration:

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the Postfilter test directory (\WINCE500\SUPPORT\TESTS\PF).
- Enter "***set WINCEREL=1***" on the command prompt and hit <return>. This will copy the generated ".dll" to the flat release directory.
- Enter "***build –c***" at the prompt, and then press <return>. The pftest.dll file will be located in the $(_FLATRELEASEDIR) directory.
- Copy all test data files (CIF_six_frames_h264.bs, CIF_six_frames_h264.qp, CIF_six_frames_h264.yuv, H264RefOutput.yuv, QCIF_twenty_frames_mpeg4.qp, QCIF_twenty_frames_mpeg4.yuv) from \WINCE500\SUPPORT\TESTS\PF to the $(_FLATRELEASEDIR) directory.
- Copy Tux.exe, Kato.dll, Tooltalk.dll from Windows CE CETK to the $(_FLATRELEASEDIR) directory.

To build the Postfilter unit test in Windows CE 6.0, complete the following steps.

Build an OS image for the desired configuration:

- Within Microsoft Visual Studio, go to the **Build** menu option and select the **Open Release Directory in Build Window** menu option. This will open a DOS prompt.
- Change to the Postfilter test directory (\WINCE600\SUPPORT\MX31\TESTS\PF or \WINCE600\SUPPORT\MX32\TESTS\PF).
- Enter "***set WINCEREL=1***" on the command prompt and hit <return>. This will copy the generated ".exe" to the flat release directory.
- Enter "***build –c***" at the prompt, and then press <return>. The pftest.dll file will be located in the $(_FLATRELEASEDIR) directory.
- Copy all test data files (CIF_six_frames_h264.bs, CIF_six_frames_h264.qp, CIF_six_frames_h264.yuv, H264RefOutput.yuv, QCIF_twenty_frames_mpeg4.qp, QCIF_twenty_frames_mpeg4.yuv) from \WINCE600\SUPPORT\MX31\TESTS\PF or \WINCE600\SUPPORT\MX32\TESTS\PF to the $(_FLATRELEASEDIR) directory.

23.5.3 Running the Postfilter Tests

After downloading an OS image to the board, the unit test can be executed from the target shell with the following command:

"**s *tux –o –d \release\pftest.dll"*** on Windows CE 5.0

"**s *\release\pftest.exe"***on Windows CE 6.0

23.6 Postfilter Driver API Reference

23.6.1 Postfilter Driver Functions

23.6.1.1 PFOpenHandle

This API creates a handle to the Postfilter stream driver:

```
HANDLE PFOpenHandle(
        void
);
```

Parameters

This API accepts no parameters.

Return Values

An open handle to the specified file indicates success. INVALID_HANDLE_VALUE indicates failure.

Remarks

A handle returned successfully from this function call is required in all subsequent calls to other PF API functions. Use the **PFCloseHandle** function to close the handle returned by **PFOpenHandle**.

23.6.1.2 PFCloseHandle

This API function closes a handle to the PF driver:

```
HANDLE PFCloseHandle(
        HANDLE hPF
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

Return Values

Nonzero indicates success.

Zero indicates failure.

To get extended error information, call GetLastError.

An open handle to the specified file indicates success.

Remarks

None.

23.6.1.3 PFConfigure

This API configures the Postfilter driver:

```
void PFConfigure(
```

```
        HANDLE hPF,
        pPfConfigData pConfigData
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

pConfigData

[in] An object of the ***pfConfigData*** structure.

Return Values

None.

Remarks

This function performs configuration steps that are required before starting a Post-Filtering operation. Calling **PFStart** without previously calling **PFConfigure** will result in an error.

23.6.1.4 PFStart

This API function starts a Postfilter operation.

```
void PFStart(
        HANDLE hPF,
        pPfStartParams pStartParms
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

pStartParms

[in] An object of the ***pfStartParams*** structure. For H.264 Postfilter mode, no output buffer is required, as the input buffer is used for input and output.

Return Values

None.

Remarks

Calling **PFStart** without previously calling **PFConfigure** will result in an error.

Completion of the Postfilter operation is signalled through a named event using the name PF_EOF_EVENT_NAME. A user can call **CreateEvent** and **WaitForSingleObject** to create and wait on the PostFilter end-of-frame event.

23.6.1.5 PFResume

This API function resumes an H.264 deblocking operation that was previously started with a pause row specified. A new pause row may be specified, or the operation may be allowed to run to completion.

```
DWORD PFResume(
```

```
        HANDLE hPF,
        UINT32 h264_pause_row
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

h264_pause_row

[in] Integer indicating the Y row at which to pause the operation. Should be set to 0 to disable an additional pause.

Return Values

If successful, PF_SUCCESS.

If failure, one of the following:

PF_ERR_NOT_RUNNING – The Postfilter operation is not running.

PF_ERR_PAUSE_NOT_ENABLED – The H.264 pause is not enabled.

PF_ERR_INVALID_PARAMETER – The pause row parameter is not in a valid range.

Remarks

Calling **PFResume** without previously calling **PFStart** with the pause row enabled will result in an error.

23.6.1.6 PFAllocPhysMem

This API function allocates physically contiguous memory.

```
BOOL PFAllocPhysMem(
        HANDLE hPF,
        UINT32 size,
        pPfAllocMemoryParams pBitsStreamBufMemParams
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

size

[in] Number of bytes to be allocated.

pBitsStreamBufMemParams

[out] Pointer to a pPfAllocMemoryParams struct that stores the memory parameters of the memory allocation.

Return Values

If successful, return TRUE.

If failure, return FALSE.

Remarks

This Postfilter API is for Windows CE 6.0 only.

23.6.1.7 PFFreePhysMem

This API function frees the memory allocated by **PFAllocPhysMem**.

```
BOOL PFFreePhysMem(
        HANDLE hPF,
        pPfAllocMemoryParams bitsStreamBufMemParams
);
```

Parameters

hPF

[in] Handle to the PF driver returned by **PFOpenHandle** API.

bitsStreamBufMemParams

[in] Virtual memory address parameters returned by **PFAllocPhysMem** API.

Return Values

If successful, return TRUE.

If failure, return FALSE.

Remarks

This Postfilter API is for Windows CE 6.0 only.

23.6.2 PF Driver Enumerations

23.6.2.1 pfMode

Enumeration of Postfilter operation modes.

```
typedef enum pfModeEnum
{
    pfType_Disabled,           // No post-filtering
    pfType_MPEG4Deblock,       // MPEG4 Deblock only
    pfType_MPEG4Dering,        // MPEG4 Dering only
    pfType_MPEG4DeblockDering, // MPEG4 Deblock and Dering
    pfType_H264Deblock,        // H.264 Deblock
} pfMode;
```

Elements

pfType_Disabled

Postfiltering disabled.

pfType_MPEG4Deblock

Postfiltering operation is MPEG4 Deblock only.

pfType_MPEG4Dering

Postfiltering operation is MPEG4 Dering only.

pfType_MPEG4DeblockDering

Postfiltering operation is MPEG4 Deblock and Dering.

pfType_H264Deblock

Postfiltering operation is H.264 Deblock.

Remarks

None.

23.6.3 PF Driver Structures

23.6.3.1 pfBuffer

Structure to describe the YUV buffers used in Postfilter operations.

```
typedef struct pfBufferStruct
{
    int      size;
    UINT32  *yBufPtr;
    UINT32   uOffset;
    UINT32   vOffset;
} pfBuffer, *pPfBuffer;
```

Members

size

Size of the allocated buffer, in bytes.

yBufPtr

Pointer to the start of the Y buffer.

uBufOffset

Offset, in bytes, of the U buffer, relative to the start of the Y buffer. If set to 0, a default calculation will be made based on the height and stride of the frame.

vBufOffset

Offset, in bytes, of the V buffer, relative to the start of the Y buffer. If set to 0, a default calculation will be made based on the height and stride of the frame.

23.6.3.2 pfConfigData

Structure used to configure the Postfilter driver for an operation.

```
typedef struct pfConfigDataStruct
{
    pfMode mode;
    pfFrameSize frameSize;
    UINT32 frameStride;
    UINT32 *qpBuf;
```

```
    UINT32 qpSize;
} pfConfigData, *pPfConfigData;
```

Members

mode

The Post-filter operation desired.

frameSize

The dimensions of the frame for Postfiltering.

frameStride

The stride of the frame, in bytes.

qpBuf

A pointer to a buffer containing, sequentially, the quantization parameter (QP) and boundary strength (BS) data for the Postfilter operation.

qpSize

The size of the buffer containing the QP and BS data.

Remarks

For H.264 deblocking, there is one 32-bit quantization parameter word for each 16x16 pixel macroblock. Additionally, there is one 8-bit boundary strength word for each 4x4 pixel block.

For MPEG4 deblocking and deringing, there is one 8-bit quantization parameter word for each 16x16 pixel macroblock. No boundary strength data is needed for MPEG4 deblocking and deringing.

23.6.3.3 pfFrameSize

Structure for the Postfiltering frame size.

```
typedef struct pfFrameSizeStruct {
    UINT16 width;
    UINT16 height;
} pfFrameSize, *pPfFrameSize;
```

Members

width

Frame width, in pixels.

height

Frame height, in pixels.

23.6.3.4 pfStartParams

This structure is used in calls to **PFStart** to provide information needed to start the Postfilter operation.

```
typedef struct PfStartParamsStruct
```

```
{
    pPfBuffer in;
    pPfBuffer out;
    UINT32 h264_pause_row;
} pfStartParams, *pPfStartParams;
```

Members

in

Pointer to the input buffer.

out

Pointer to the output buffer.

h264_pause_row

Row to pause at for H.264 mode. Set to 0 to disable pause. For more information, refer to the Postfilter Flow Control section of the MX31/MX32 hardware specification.

23.6.3.5 pfAllocMemoryParams

This structure is used in calls to **PFAllocPhysMem/PFFreePhysMem** to allocate/free physically contiguous memory in Windows CE 6.0.

```
typedef struct pfFrameSizeStruct {
    UINT physAddr;
    UINT userVirtAddr;
    UINT driverVirtAddr;
    UINT size;
} pfAllocMemoryParams, *pPfAllocMemoryParams;
```

Members

physAddr

A physical memory address of the memory allocation.

userVirtAddr

A virtual memory address of the memory allocation.

driverVirtAddr

A virtual memory address to be used in the kernel mode inside the driver.

size

A memory size to be allocated.

Chapter 24
Power Management IC (PMIC)

This chapter provides the information that you need to:

- Develop device drivers that interface directly to the hardware components provided by Freescale Semiconductor's power management ICs (PMICs). The PMIC that is specifically referenced in this document is the MC13783.
- Develop applications that make use of the special hardware capabilities that are provided by the PMIC (e.g., audio I/O, USB On-The-Go connectivity, etc.).

This chapter fully describes the API provided by Freescale which allows complete access to the full functionality of the PMICs.

This document is intended for device driver and application developers who need to understand and gain access to the functionality provided by the PMICs.

24.1 PMIC Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	..\CSP\ARM\FREESCALE\PMIC\MC13783\PDK ..\CSP\ARM\FREESCALE\PMIC\MC13783\SDK
CSP Static Library	pmicPdkCsp_MC13783.lib pmicSdkCsp_mc13783.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\PMIC\MC13783\PDK
Import Library	N/A
Driver DLL	pmicPdk_MC13783.dll
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PMIC_MC13783=1

24.2 Requirements

The PMIC device driver framework for Windows CE is a stream interface driver and a SDK DLL. 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 PMIC Stream Interface driver controls the PMIC hardware directly via the SPI bus. The Stream Interface driver provides an IOCTL interface for SDK DLLs. The SDK DLL provide APIs for Windows CE drivers and applications.

The API covers the PMIC functionality of the following areas:

- Register Access
- Audio
- Tri-Color LED
- Battery
- Regulators
- Keys (Power, PTT)
- ADC /Touch
- Backlight (Keyboard, LCD)
- End of Life comparator
- Power Fail
- Battery Charger
- USB/RS232 transceiver
- Vibrator
- GPIO
- CE Bus

24.2.1 PMIC API Framework

The API framework and the APIs defined in this document are intended to be reused for all Freescale power management ICs. The current implementation of the APIs supports the MC13783 power management IC.

The APIs presented in this document were developed to provide a unified interface to all of the functions and features provided by the power management ICs. When a specific function exists on all of the power management ICs, then the API will behave identically (e.g., selecting the USB connectivity operating mode).

A device driver and API framework for Wince has already been implemented for the MC13783 PMIC. The existing MC13783 framework will be reused as-is but the APIs will be redefined so that a single unified set of APIs can be used to access the features and functions provided by both the MC13783 PMICs. The key objectives and benefits of this new generic PMIC API are as follows:

- Provide the ability to easily accommodate additional PMICs in the future (perhaps with additional features or configuration options) without breaking existing software.

- Provide a uniform API for accessing all Freescale PMICs so that device drivers and applications can be ported to different hardware platforms with little or no changes.
- Provide the ability to access all of the underlying hardware capabilities provided by a specific PMIC. Of course, any software that makes use of PMIC-specific functions or configurations will only work if the appropriate PMIC hardware is actually available. However, the software should still provide appropriate error codes and “fail gracefully” if it is used on a platform for which the requested function or configuration is not supported.

24.3 Hardware Operation

Refer to the MC13783 (MC13783) document for details on the MC13783 PMIC.

24.4 Conflicts with Other SoC Peripherals

24.4.1 MX31, MX32 Peripheral Conflicts

There are no MX31 pin conflicts on the MX31 ADS. The CSPI2 is used to communicate with the MC13783 PMIC on the MX31 platform, the CSPI2 signals are selected in the IOMUX.

24.5 Software Operation

24.5.1 Configuring the PMIC

The PMIC modules can be used by applications or device drivers Ex: Battery API’s of PMIC will be used by the Battery driver.

Configuring the PMIC port for communications involves some basic operations:

A handle to the desired PMIC port must be opened prior to accessing the module registers. This handle is required to call the **DeviceIoControl** function. The function parameters include the PMIC port handle, appropriate IOCTL code, and other input and output parameters.

24.5.2 Creating a Handle to the PMIC

Before calling any of the PMIC API’s make sure that PMIC device is attached by calling the **CreateFile** function which opens a file and it returns a handle that can be used to access the MC13783 hardware. If MC13783 hardware does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the PMIC:

Insert a colon after the PMI1 port for the first parameter, *lpFileName*.

For example, specify PMI1: as the PMIC port.

Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to a PMIC port are supported by the driver.

Specify OPEN_EXISTING in the *dwCreationDisposition* parameter.

This flag is required.

Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

The following code example shows how to open a PMIC port.

```
hPMI = CreateFile(TEXT("PMI1:"), GENERIC_READ | GENERIC_WRITE,  access (read-write) mode
            FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,  sharing mode
            FILE_FLAG_RANDOM_ACCESS, NULL);  security attributes
        if ((hPMI == NULL) || (hPMI == INVALID_HANDLE_VALUE))
        {
            ERRORMSG(1, (_T("Failed in create File()\r\n")));
        }
```

NOTE

All the above specified steps are performed when PMIC devices attach is called. If hPMI handle is null then perform the above steps.

24.5.3 Write Operations

The PMIC driver does not provide an interface to write through the PMIC_Write (stream write) function in PMIC driver and PMIC_Write is a stub function and returns success always.

24.5.4 Read Operations

Like the write operation, the PMIC driver does not provides no facilitate for reading through the PMIC_Read function in PMIC driver, this is a stub function and returns success always.

24.5.5 Closing the Handle to the PMIC

Call the **CloseHandle** function to close a handle to the PMIC when an application is done using it.

CloseHandle has one parameter, which the handle is returned by the CreateFile function call that opened the PMIC port.

24.5.6 Power Management

The primary method for limiting power consumption in the PMIC module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. PMIC module clock is enabled whenever any of the PMIC registers needs to be accessed and then disabled once it is done.

24.5.6.1 PowerUp

This function is not implemented for the PMIC driver.

24.5.6.2 PowerDown

This function is not implemented for the PMIC driver.

24.5.6.3 IOCTL_POWER_CAPABILITIES

We advertise the power management capabilities with power manager through this IOCTL. The PMIC module supports only two power states: D0 and D4.

24.5.6.4 IOCTL_POWER_SET

This IOCTL requests a change from one device power state to another. D0 and D4 are the only two supported **CEDEVICE_POWER_STATE** in PMIC driver. Any request that is not D0 is changed to a D4 request & will result in the system entering into suspend state, while for a value of D0 the system will again be resumed.

24.5.6.5 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 will not generally issue an **IOCTL_POWER_GET** call to the device unless an application calls **GetDevicePower** with the POWER_FORCE flag set.

24.6 PMIC Registry Settings

There are no registry settings that need to be modified to use the PMIC API's.

24.7 Unit Test

The PMIC CETK test cases verify the functionality of the various PMIC components.

24.7.1 Unit Test Hardware

The ADS and the MC13783 PMIC board are required.

24.7.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
PMICtest.dll	Test .dll file

24.7.3 Building the PMIC Tests

In order to build the PMIC tests, complete the following steps:

Build an OS image for the desired configuration.

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
- Change to the PMIC Tests directory. (\WINCE500\SUPPORT\TESTS\PMIC)
- Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
- Enter the build command (*build -c*) at the prompt and press return.

After the build completes, the pmictest.dll file will be located in the $(_FLATRELEASEDIR) directory.

24.7.4 Running the PMIC Tests

The command line for running the PMIC tests is ***tux –o –d pmictest***. You can even provide an additional option –f if you wish to redirect the test results on to a file. The PMIC tests do not contain any test specific command line options.

The following table describes the test cases contained in the PMIC tests.

#	Test name	Description
1	PMIC Register Access	This test does read/write verification of IMR register on the PMIC.
2	BatteryTest	This test checks the PMIC Battery interface. It will read and verify the charger parameters.
4	Voltage Regulator	This test verifies the voltage regulator and switchers on the PMIC. These include: testing on and off various voltage regulators, reading the voltage level, setting all possible voltage levels (which can be verified on test points, and matched with the message displayed), verifying the vibrator activity, setting various values in the switch mode voltage regulator.
5	Power Control	This test verifies the power control functionality on the PMIC.
6	AdcGetOneSample	This test gets one sample from each of 16 channels by requesting the driver to sample one channel at a time.
7	AdcGet8Samples	This test gets 8 samples from each of 16 channels.
8	AdcGetMultiChannelSamples	This test gets one sample from each of 16 channels by requesting the driver to sample all 16 channels at once.
9	AdcGetHandsetCurrent	This test gets samples of handset current.
10	AdcTouchRead	This test gets 3 (x,y) coordinates from the touch screen.
11	Backlight On	This test requires user input to identify a passing condition. (Uses LEDs on MC13783 bd.)
12	Backlight Current Control Mode	This test configures the backlight driver for current mode and steps through different current levels. (Uses LEDs on MC13783 bd.)
13	Backlight Triode Mode	This test configures the backlight for triode mode and steps through different current levels. (Uses LEDs on MC13783 bd.)
14	Backlight Off	This test requires user input to identify a passing condition. (Uses LEDs on MC13783 bd.)

24.8 PMIC Reference

24.8.1 PMIC Driver IOCTLS

This section consists of descriptions for the PMIC I/O control codes (IOCTLs). These IOCTLs are used in calls to DeviceIoControl to issue commands to the PMIC device modules. Only relevant parameters for the IOCTL have a description provided. These IOCTL's are used with in the API's developed for specific modules of the PMIC device. Most of the IOCTL's will be explained in the specific sections where ever they are more relevant.

24.8.1.1 PMIC_IOCTL_LLA_READ_REG

This **DeviceIoControl** request reads the register content.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

[out] Long pointer to a buffer that receives the output data for the operation. Set to NULL if the dwIoControlCode parameter specifies an operation that does not produce output data.

PMIC_IOCTL_LLA_WRITE_REG

This **DeviceIoControl** request writes the data to the said register of the PMIC device.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

pointer to data which needs to be written to the said register.

24.8.1.2 PMIC_IOCTL_LLA_INT_REGISTER

This **DeviceIoControl** is used to register interrupt.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

pointer to event name and interrupt id.

Code example:

```
param.int_id = int_id;
param.event_name = event_name;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_REGISTER, &param,
      sizeof(param), NULL, 0, NULL, NULL);
```

24.8.1.3 PMIC_IOCTL_LLA_INT_DEREGISTER

This **DeviceIoControl** is used to deregister pmic interrupt.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

null.

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_DEREGISTER, &param,
      sizeof(param), NULL, 0, NULL, NULL);
```

24.8.1.4 PMIC_IOCTL_LLA_INT_COMPLETE

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

pointer to interrupt id.

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,
      sizeof(param), NULL, 0, NULL, NULL);
```

24.8.1.5 PMIC_IOCTL_LLA_INT_ENABLE

This IOCTL is used to enable the interrupt.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

pointer to interrupt id.

Code example :

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,
      sizeof(param), NULL, 0, NULL, NULL);
```

24.8.1.6 PMIC_IOCTL_LLA_INT_DISABLE

This IOCTL is used to disable the interrupt.

Parameters *hPMI*

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

lpInBuffer

index of the register.

lpOutBuffer

pointer to interrupt id.

Code example :

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,
      sizeof(param), NULL, 0, NULL, NULL);
```

24.9 Interrupt Handling

24.9.1 Interrupt handling Overview

The PMIC has interrupt generation capability to inform the CPU when events occur. This is signaled to the processors driving the primary SPI and secondary SPI busses via the PRIINT and SECINT lines, respectively. There is only one interrupt line connected to each processor, so the kernel can only know that there is an interrupt from the PMIC, but without knowing exactly which module generated the interrupt.

There is one PMIC Interrupt Service Thread (IST) to handle all interrupts from the PMIC. The PMIC IST will be invoked by the kernel once the kernel receives an interrupt from the PMIC.

This IST will first query the PMIC to determine the source of the interrupt. The IST maintains a table to track if an interrupt has been registered by a driver or application. If the interrupt is registered, the IST will then set a predefined event.

For any drivers and applications that need notification of an interrupt, they must register the interrupt and wait for the event. They also need to reset the event after handling the event.

24.9.2 Interrupt Events

Drivers or applications that wish to monitor an interrupt should create a named event for each interrupt. The event name is passed to PMIC driver when registering the interrupt.

The PMIC IST will trigger the event when the corresponding interrupt occurs.

24.9.2.1 PMIC Interrupt Events

The table below shows the events and corresponding MC13783 interrupts.

Table 24-1. PMIC Interrupts

PMIC Interrupt	Description
ADCDONEI	ADC has finished requested conversions
ADCBISDONEI	ADCBIS has finished requested conversions
TSI	Touch screen wakeup
WHI	A/D word read in ADC digital comparison mode exceeding the high limit
WLI	A/D word read in ADC digital comparison mode reading below the low limit
CHGDETI	Charger attach and removal
CHGOVI	Charger over-voltage detection
CHGREVI	Charger path reverse current
CHGSHORTI	Charger path short circuit
CCCVI	BP regulator current or voltage regulation. Indicates that the charger has switched its mode from CC to CV or from CV to CC. Charger removal does not trigger this interrupt.
CHGCURRI	Charge current has dropped below threshold
BPONI	BP turn on threshold detection
LOBATLI	End of lift/low battery detection
LOBATHI	Low battery warning
USBI	USB VBUS detection
IDI	USB ID line detection
SE1I	Single ended 1 detection
CKDETI	Carkit detection
1HZI	1HZ timetick
TODAI	Time of day alarm. Triggered when TOD counter is equal to the value is TODA and the DAY counter is equal to the value in DAYA.
ONOFD1I	ON1B event. Connection for a power on/off button.
ONOFD2I	ON2B event. Connection for an accessory power on/off button

Table 24-1. PMIC Interrupts (continued)

PMIC Interrupt	Description
ONOFD3I	ON3B event. Connection for a third power on/off button.
SYSRSTI	Indicates system reset has occurred
RTCRSTI	Indicates RTC reset has occurred
PCI	Indicates power cut has occurred
WARMI	Warm start event. Indicates the application powered up from user off mode.
MEMHLDI	Memory hold event. Indicates the application powered up from memory hold mode.
PWRRDYI	Power gate and DVS power ready
THWARNLI	Thermal warning lower threshold
THWARNHI	Thermal warning higher threshold
CLKI	Clock source change
SEMAFI	Semaphore
MC2BI	Microphone bias 2 detect
HSDETI	Headset attach
HSLI	Stereo headset detect
ALSPTHI	Thermal shutdown Alsp. Maximum allowable junction temperature within Alsp is reached.
AHSSHORTI	Short circuit on Ahs outputs

24.9.2.2 Interrupt Data Structures

```
typedef enum _PMIC_MC13783_INT_ID {
    PMIC_MC13783_INT_ADCDONEI = 0,
    PMIC_MC13783_INT_ADCBISDONEI = 1,
    PMIC_MC13783_INT_TSI = 2,
    PMIC_MC13783_INT_WHI = 3,
    PMIC_MC13783_INT_WLI = 4,
    PMIC_MC13783_INT_CHGDETI = 6,
    PMIC_MC13783_INT_CHGOVI = 7,
    PMIC_MC13783_INT_CHGREVI = 8,
    PMIC_MC13783_INT_CHGSHORTI = 9,
    PMIC_MC13783_INT_CCCVI = 10,
    PMIC_MC13783_INT_CHGCURRI = 11,
    PMIC_MC13783_INT_BPONI = 12,
    PMIC_MC13783_INT_LOBATLI = 13,
    PMIC_MC13783_INT_LOBATHI = 14,
    PMIC_MC13783_INT_USBI = 16,
    PMIC_MC13783_INT_IDI = 19,
    PMIC_MC13783_INT_SE1I = 21,
    PMIC_MC13783_INT_CKDETI = 22,
    PMIC_MC13783_INT_1HZI = 32,
    PMIC_MC13783_INT_TODAI = 33,
    PMIC_MC13783_INT_ONOFD1I = 35,
    PMIC_MC13783_INT_ONOFD2I = 36,
    PMIC_MC13783_INT_ONOFD3I = 37,
```

```
    PMIC_MC13783_INT_SYSRSTI = 38,
    PMIC_MC13783_INT_RTCRSTI = 39,
    PMIC_MC13783_INT_PCI = 40,
    PMIC_MC13783_INT_WARMI = 41,
    PMIC_MC13783_INT_MEMHLDI = 42,
    PMIC_MC13783_INT_PWRRDYI = 43,
    PMIC_MC13783_INT_THWARNLI = 44,
    PMIC_MC13783_INT_THWARNHI = 45,
    PMIC_MC13783_INT_CLKI = 46,
    PMIC_MC13783_INT_SEMAFI = 47,
    PMIC_MC13783_INT_MC2BI = 49,
    PMIC_MC13783_INT_HSDETI = 50,
    PMIC_MC13783_INT_HSLI = 51,
    PMIC_MC13783_INT_ALSPTHI = 52,
    PMIC_MC13783_INT_AHSSHORTI = 53,
    PMIC_INT_MAX_ID
} PMIC_MC13783_INT_ID;
```

24.9.2.3 Functions

24.9.2.3.1 PmicInterruptRegister

PmicInterruptRegister function registers an interrupt so that the interrupt event will be signaled when the interrupt occurs.

All PMIC interrupts are masked at the initialization. A driver or an application must register the interrupt if the interrupt is to be enabled.

Prototype *PMIC_STATUS PmicInterruptRegister(PMIC_INT_ID int_id, LPTSTR name);*

Parameters: *int_id*

[in] The interrupt to be registered

name

[in] The event name

Return Value Status code

Remarks:

In this function the PMIC_IOCTL_LLA_INT_REGISTER IOCTL code is used and below is the code example.

```
 param.int_id = int_id;
param.event_name = event_name;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_REGISTER, &param,
          sizeof(param), NULL, 0, NULL, NULL);
if (ret)
{
    return PMIC_SUCCESS;
}
else
{
    return PMIC_ERROR;
}
```

24.9.2.3.2 PmicInterruptDeregister

PmicInterruptDeregister function deregisters an interrupt. If an interrupt is not registered by any driver or application, it will be masked.

Prototype *PMIC_STATUS PmicInterruptDeregister(PMIC_INT_ID int_id);*

Parameters *int_id*

[in] The interrupt to be deregistered

Return Value Status code

Remarks

In this function the PMIC_IOCTL_LLA_INT_DEREGISTE call is used and below is the code example

```
param = int_id;

ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_DEREGISTER, &param,
          sizeof(param), NULL, 0, NULL, NULL);
if (ret)
{
    return PMIC_SUCCESS;
}
else
{
    return PMIC_ERROR;
}
```

24.9.2.3.3 PmicInterruptHandlingComplete

PmicInterruptHandlingComplete function notifies the PMIC stream interface driver completion of an interrupt handling, so that the stream interface driver can enable that interrupt again.

Prototype *PMIC_STATUS PmicInterruptHandlingComplete(PMIC_INT_ID int_id);*

Parameters int_id

[in] The interrupt index.

Return Value Status code

Remarks

In this function the PMIC_IOCTL_LLA_INT_COMPLETE call is used and below is the code example

```
param = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,
          sizeof(param), NULL, 0, NULL, NULL);
if (ret)
  {
     return PMIC_SUCCESS;
  }
  else
  {
     return PMIC_ERROR;
   }
```

24.9.2.3.4 PmicInterruptDisable

PmicInterruptDisable function temporarily disables an interrupt. The interrupt is still registered. The driver or application can enable the interrupt again by calling PmicInterruptEnable().

Prototype *PMIC_STATUS PmicInterruptDisable(PMIC_INT_ID int_id);*

Parameters *int_id*

[in] The interrupt index.

Return Value Status code

Remarks

In this function the PMIC_IOCTL_LLA_INT_DISABLE call is used and below is the code example

```
param = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_DISABLE, &param,
          sizeof(param), NULL, 0, NULL, NULL);
if (ret)
{
    return PMIC_SUCCESS;
}
else
{
    return PMIC_ERROR;
}
```

24.9.2.3.5 PmicInterruptEnable

PmicInterruptEnable function re-enable an interrupt.

Prototype *PMIC_STATUS PmicInterruptEnable(PMIC_INT_ID int_id);*

Parameters *int_id*

[in] The interrupt index.

Return Value Status code

Remarks

In this function the PMIC_IOCTL_LLA_INT_ENABLE call is used and below is the code example

```
 param = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_ENABLE, &param,
          sizeof(param), NULL, 0, NULL, NULL);
if (ret)
{
    return PMIC_SUCCESS;
}
else
{
    return PMIC_ERROR;
}
```

Code example of registering PMIC pen down interrupts.

```
if (PmicInterruptRegister(PMIC_MC13783_INT_TSI, _T("EVENT_TS"))
    != PMIC_SUCCESS)
{
    ERRORMSG(1, (_T("PmicInterruptRegister failed\r\n")));
```

```
            goto cleanUp;
        }
```

Deregister for PMIC pen down interrupts.

```
        PmicInterruptDeregister(PMIC_MC13783_INT_TSI);
```

24.10 Register Access API

The PMIC Low Level Access API allows drivers and/or applications to read and write PMIC registers. There are some restrictions to prohibit drivers/application from accessing some registers. Interrupt registers is one example. The interrupt library functions will be in this Low Level Access DLL.

24.10.1 Functions

24.10.1.1 Read Register

This function reads a PMIC register.

Prototype

```
PMIC_STATUS
PmicRegisterRead(unsigned char index, UINT32* reg);
```

Parameters

index

[in] register index.

reg

[out] The contents of the register.

Return Value Status code

24.10.1.2 Write Register

This function writes a PMIC register.

Prototype

```
PMIC_STATUS
PmicRegisterWrite(unsigned char index, UINT32 reg, UINT32 mask);
```

Parameters

index

[in] register index.

reg

[in] data to be written.

mask

[in] bitmap mask to indicate which bits in parameter reg should be written to PMIC register.

Return Value Status code

The following code example shows how to use PmicRegisterWrite / PmicRegisterRead function to write/read to/from the PMIC module registers.

```
TESTPROCAPI PMICTest1(UINT uMsg, TPPARAM tpParam, LPFUNCTION_TABLE_ENTRY lpFTE)
{
```

```
    UINT32 reg;

     Validate that the shell wants the test to run
    if (uMsg != TPM_EXECUTE)
    {
        return TPR_NOT_HANDLED;
    }
    g_pKato->Log(LOG_COMMENT, TEXT("PMICTest1() +\r\n"));
     Read IMR
    PmicRegisterRead(1, &reg);
    g_pKato->Log(LOG_COMMENT, TEXT("Register IMR is 0X%X\r\n"), (reg & 0xFFFFFF));
    g_pKato->Log(LOG_COMMENT, TEXT("Now, try to change IMR to 0x0FF\r\n"));
    PmicRegisterWrite(1, 0x0FF, 0xFFFFFF);
    PmicRegisterRead(1, &reg);
    g_pKato->Log(LOG_COMMENT, TEXT("Register IMR is 0X%X\r\n"), (reg & 0xFFFFFF));
     Enter ISR loop
    if ((reg&0xFFFFFF) == 0xFF)
    {
        GPT_TEST_FUNCTION_EXIT();
        return TPR_PASS;
    }
    else
    {
        GPT_TEST_FUNCTION_EXIT();
        return TPR_FAIL;
    }
}
```

24.11 Power Control Reference

24.11.1 PwCtrl API

This chapter provides information about API provided by PwCtrl API DLL.

Using the following API's MC13783 Power control module can be accessed.

Module	Usage
PmicPwrctrlSetPowerCutTimer	used to set the power cut timer duration
PmicPwrctrlGetPowerCutTimer	used to get the power cut timer duration
PmicPwrctrlEnablePowerCut	used to enable the power cut
PmicPwrctrlDisablePowerCut	used to disable the power cut
PmicPwrctrlSetPowerCutCounter	used to set the power cut counter
PmicPwrctrlGetPowerCutCounter	used to get the power cut counter
PmicPwrctrlSetPowerCutMaxCounter	used to set the maximum number of power cut counter
PmicPwrctrlGetPowerCutMaxCounter	used to get the setting of maximum power cut counter
PmicPwrctrlEnableCounter	function will set PC_COUNT_EN=1
PmicPwrctrlDisableCounter	function will set PC_COUNT_EN=0
PmicPwrctrlSetMemHoldTimer	used to set the duration of memory hold timer

Module	Usage
PmicPwrctrlGetMemHoldTimer	used to get the setting of memory hold timer
PmicPwrctrlSetMemHoldTimerAllOn	used to set the duration of the memory hold timer to infinity
PmicPwrctrlClearMemHoldTimerAllOn	used to clear the infinity duration of the memory hold timer
PmicPwrctrlEnableClk32kMCU	used to enable the CLK32KMCU
PmicPwrctrlDisableClk32kMCU	used to disable the CLK32KMCU
PmicPwrctrlEnableUserOffModeWhenDelay	used to place the phone in User Off Mode after a delay
PmicPwrctrlDisableUserOffModeWhenDelay	used to set not to place the phone in User Off Mode after a delay
PmicPwrctrlSetVBKUPRegulator	used to set the VBKUP regulator
PmicPwrctrlSetVBKUPRegulatorVoltage	used to set the VBKUP regulator voltage
PmicPwrctrlEnableWarmStart	used to set the phone to transit from the ON state to the User Off state when either the USER_OFF pin is pulled high or the USER_OFF_SPI bit is set
PmicPwrctrlDisableWarmStart	This function is used to disable the warm start and set the phone to transit from the ON state to the MEMHOLD ONLY state when either the USER_OFF pin is pulled high or the USER_OFF_SPI bit is set
PmicPwrctrlEnableRegenAssig	Used enables the REGEN pin of selected voltage regulator
PmicPwrctrlDisableRegenAssig	Used disable the REGEN pin of selected voltage regulator
PmicPwrctrlGetRegenAssig	reads the REGEN pin value for said voltage regulator

24.11.2 Functions and Data Structures

```
PMIC_STATUS PmicPwrctrlSetPowerCutTimer (UINT8 duration);
PMIC_STATUS PmicPwrctrlGetPowerCutTimer (UINT8* duration);
PMIC_STATUS PmicPwrctrlEnablePowerCut (void);
PMIC_STATUS PmicPwrctrlDisablePowerCut (void);
PMIC_STATUS PmicPwrctrlSetPowerCutCounter (UINT8 counter);
PMIC_STATUS PmicPwrctrlGetPowerCutCounter (UINT8* counter);
PMIC_STATUS PmicPwrctrlSetPowerCutMaxCounter (UINT8 counter);
PMIC_STATUS PmicPwrctrlGetPowerCutMaxCounter (UINT8* counter);
PMIC_STATUS PmicPwrctrlEnableCounter(void);
PMIC_STATUS PmicPwrctrlDisableCounter (void);
PMIC_STATUS PmicPwrctrlSetMemHoldTimer (UINT8 duration);
PMIC_STATUS PmicPwrctrlGetMemHoldTimer (UINT8* duration);
PMIC_STATUS PmicPwrctrlSetMemHoldTimerAllOn (void);
PMIC_STATUS PmicPwrctrlClearMemHoldTimerAllOn (void);
PMIC_STATUS PmicPwrctrlEnableClk32kMCU (void);
PMIC_STATUS PmicPwrctrlDisableClk32kMCU (void);
PMIC_STATUS PmicPwrctrlEnableUserOffModeWhenDelay (void);
PMIC_STATUS PmicPwrctrlDisableUserOffModeWhenDelay (void);
PMIC_STATUS              PmicPwrctrlSetVBKUPRegulator              (MC13783_PWRCTRL_REG_VBKUP,
MC13783_PWRCTRL_VBKUP_MODE);
PMIC_STATUS PmicPwrctrlSetVBKUPRegulatorVoltage (MC13783_PWRCTRL_REG_VBKUP, UINT8);
PMIC_STATUS PmicPwrctrlEnableWarmStart (void);
PMIC_STATUS PmicPwrctrlDisableWarmStart (void);
```

```
PMIC_STATUS PmicPwrctrlEnableRegenAssig (t_regulator regu);
PMIC_STATUS PmicPwrctrlDisableRegenAssig (t_regulator regu);
PMIC_STATUS PmicPwrctrlGetRegenAssig (t_regulator regu, UINT8* value);

 The backup regulators VBKUP1 and VBKUP2 provide two independent low power
 supplies during memory hold, user off and power cut operation.
typedef enum _MC13783_PWRCTRL_REG_VBKUP{
   VBKUP1,
   VBKUP2,
} MC13783_PWRCTRL_REG_VBKUP;

typedef enum _MC13783_PWRCTRL_VBKUP_MODE{
   VBKUP_MODE1,    Backup Regulator Off in Non Power Cut Modes and Off in Power Cut Modes
   VBKUP_MODE2,    Backup Regulator Off in Non Power Cut Modes and On in Power Cut Modes
   VBKUP_MODE3,    Backup Regulator On in Non Power Cut Modes and Off in Power Cut Modes
   VBKUP_MODE4,    Backup Regulator On in Non Power Cut Modes and On in Power Cut Modes
} MC13783_PWRCTRL_VBKUP_MODE;

/*!
 * This enumeration define all regulator enabled by regen
 */
typedef enum {
        /*!
         * VAudio
         */
        REGU_VAUDIO=0,
        /*!
         * VIOHI
         */
        REGU_VIOHI,
        /*!
         * VIOLO
         */
        REGU_VIOLO,
        /*!
         * VDIG
         */
        REGU_VDIG,
        /*!
         * VGEN
         */
        REGU_VGEN,
        /*!
         * VRFDIG
         */
        REGU_VRFDIG, /*5*/
        /*!
         * VRFREF
         */
        REGU_VRFREF,
        /*!
         * VRFCP
```

```
 */
REGU_VRFCP,
/*!
 * VSIM
 */
REGU_VSIM,
/*!
 * VESIM
 */
REGU_VESIM,
/*!
 * VCAM
 */
REGU_VCAM, /*10*/
/*!
 * VRFBG
 */
REGU_VRFBG,
/*!
 * VVIB
 */
REGU_VVIB,
/*!
 * VRF1
 */
REGU_VRF1,
/*!
 * VRF2
 */
REGU_VRF2,
/*!
 * VMMC1
 */
REGU_VMMC1,
/*!
 * VMMC2
 */
REGU_VMMC2,
/*!
 * GPO1
 */
REGU_GPO1,
/*!
 * GPP2
 */
REGU_GPO2,
/*!
 * GPO3
 */
REGU_GPO3,
/*!
 * GPO4
```

```
         */
        REGU_GPO4,
        /*!
         * REGU_NUMBER
         */
        REGU_NUMBER,
} t_regulator;
/*!
 * This tab define bit for regen of all regulator
 */
int   REGULATOR_REGEN_BIT[REGU_NUMBER]={
        0, /* VAUDIO */
        1, /* VIOHI  */
        2, /* VIOLO  */
        3, /* VDIG   */
        4, /* VGEN   */
        5, /* VRFDIG */
        6, /* VRFREF */
        7, /* VRFCP  */
        -1,  /* VSIM   */
        -1,  /* VESIM  */
        8, /* VCAM   */
        9, /* VRFBG  */
        -1,  /* VVIB   */
        10, /* VRF1   */
        11, /* VRF2   */
        12, /* VMMC1  */
        13, /* VMMC2  */
        16, /* VGPO1  */
        17, /* VGPO2  */
        18, /* VGPO3  */
        19, /* VGPO4  */
};
```

24.11.2.1 PmicPwrctrlSetPowerCutTimer

Prototype *PMIC_STATUS PmicPwrctrlSetPowerCutTimer (UINT8 duration);*

This function is used to set the power cut timer duration.

Parameters: *duration [in]*

The value to set to power cut timer register, it's from 0 to 255.
The timer will be set to a duration of 0 to 31.875 seconds, in 125 ms increments.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks

24.11.2.2 PmicPwrctrlGetPowerCutTimer

Prototype *PMIC_STATUS PmicPwrctrlGetPowerCutTimer (UINT8* duration);*

Parameters: *duration [out]*

The duration to set to power cut timer

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure.

24.11.2.3 PmicPwrctrlEnablePowerCut

Prototype *PMIC_STATUS PmicPwrctrlEnablePowerCut (void);*

This function is used to enable the power cut.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure.

24.11.2.4 PmicPwrctrlDisablePowerCut

Prototype *PMIC_STATUS PmicPwrctrlDisablePowerCut (void)*

This function is used to disable the power cut.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.5 PmicPwrctrlSetPowerCutCounter

Prototype *PMIC_STATUS PmicPwrctrlSetPowerCutCounter (UINT8 counter);*

This function is used to set the power cut counter.

Parameters: *counter [in]*

The counter number value to be set to the register. It's value from 0 to 15. The power cut counter is a 4 bit counter that keeps track of the number of rising edges of the UV_TIMER (power cut events) that have occurred since the counter was last initialized.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.6 PmicPwrctrlGetPowerCutCounter

Prototype *PMIC_STATUS PmicPwrctrlGetPowerCutCounter (UINT8* counter);*

This function is used to get the power cut counter.

Parameters: *counter [out]*

to get the counter number

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.7 PmicPwrctrlSetPowerCutMaxCounter

Prototype `PMIC_STATUS PmicPwrctrlSetPowerCutMaxCounter (UINT8 counter);`

This function is used to set the maximum number of power cut counter.

Parameters: `counter [in]`

Maximum counter number to set. It's value from 0 to 15. The power cut register provides a method for disabling power cuts if this situation manifests itself.

If PC_COUNT >= PC_MAX_COUNT, then the number of resets that have occurred since the power cut counter was last initialized exceeds the established limit, and power cuts will be disabled.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.8 PmicPwrctrlGetPowerCutMaxCounter

Prototype `PMIC_STATUS PmicPwrctrlGetPowerCutMaxCounter (UINT8* counter);`

This function is used to get the setting of maximum power cut counter.

Parameters: `counter [out]`

To get the maximum counter number

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.9 PmicPwrctrlEnableCounter

Prototype `PMIC_STATUS PmicPwrctrlEnableCounter(void);`

The power cut register provides a method for disabling power cuts if this situation manifests itself. If PC_COUNT >= PC_MAX_COUNT, then the number of resets that have occurred since the power cut counter was last initialized exceeds the established limit, and power cuts will be disabled.

This function can be disabled by setting PC_COUNT_EN=0. In this case, each power cut event will increment the power cut counter, but power cut coverage will not be disabled, even if PC_COUNT exceeds PC_MAX_COUNT.

This PmicPwrctrlEnableCounter function will set PC_COUNT_EN=1.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.10 PmicPwrctrlDisableCounter

Prototype `PMIC_STATUS PmicPwrctrlDisableCounter (void);`

The power cut register provides a method for disabling power cuts if this situation manifests itself. If PC_COUNT >= PC_MAX_COUNT, then the number of resets that have occurred since the power cut counter was last initialized exceeds the established limit, and power cuts will be disabled.

This function can be disabled by setting PC_COUNT_EN=0. In this case, each power cut event will increment the power cut counter, but power cut coverage will not be disabled, even if PC_COUNT exceeds PC_MAX_COUNT. This PmicPwrctrlEnableCounter function will set PC_COUNT_EN=0.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.11 PmicPwrctrlSetMemHoldTimer

Prototype `PMIC_STATUS PmicPwrctrlSetMemHoldTimer (UINT8 duration);`

This function is used to set the duration of memory hold timer.

Parameters: `duration [in]`

The value to set to memory hold timer register. It's from 0 to 15. The resolution of the memory hold timer is 32 seconds for a maximum duration of 512 seconds.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.12 PmicPwrctrlGetMemHoldTimer

Prototype `PMIC_STATUS PmicPwrctrlGetMemHoldTimer (UINT8* duration);`

This function is used to get the setting of memory hold timer

Parameters: `duration [out]`

To get the duration of the timer

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.13 PmicPwrctrlSetMemHoldTimerAllOn

Prototype `PMIC_STATUS PmicPwrctrlSetMemHoldTimerAllOn (void);`

This function is used to set the duration of the memory hold timer to infinity

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.14 PmicPwrctrlClearMemHoldTimerAllOn

Prototype `PMIC_STATUS PmicPwrctrlClearMemHoldTimerAllOn (void);`

This function is used to clear the infinity duration of the memory hold timer

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.15 PmicPwrctrlEnableClk32kMCU

Prototype `PMIC_STATUS PmicPwrctrlEnableClk32kMCU (void);`

This function is used to enable the CLK32KMCU

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.16 PmicPwrctrlDisableClk32kMCU

Prototype `PMIC_STATUS PmicPwrctrlDisableClk32kMCU (void);`

This function is used to disable the CLK32KMCU

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.17 PmicPwrctrlEnableUserOffModeWhenDelay

Prototype `PMIC_STATUS PmicPwrctrlEnableUserOffModeWhenDelay (void);`

This function is used to place the phone in User Off Mode after a delay.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.18 PmicPwrctrlDisableUserOffModeWhenDelay

Prototype `PMIC_STATUS PmicPwrctrlDisableUserOffModeWhenDelay (void);`

This function is used to set not to place the phone in User Off Mode after a delay.

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.19 PmicPwrctrlSetVBKUPRegulator

Prototype `PMIC_STATUS PmicPwrctrlSetVBKUPRegulator (MC13783_PWRCTRL_REG_VBKUP reg, MC13783_PWRCTRL_VBKUP_MODE mode);`

This function is used to set the VBKUP regulator

Parameters: `reg [in]`

the backup regulator to set

`mode [in]`

the mode to set to backup regulator

VBKUP_MODE1 - VBKUPxEN = 0, VBKUPxAUTO = 0

Backup Regulator Off in Non Power Cut Modes and Off in Power Cut Modes

VBKUP_MODE2 - VBKUPxEN = 0, VBKUPxAUTO = 1

Backup Regulator Off in Non Power Cut Modes and On in Power Cut Modes

VBKUP_MODE3 - VBKUPxEN = 1, VBKUPxAUTO = 0

Backup Regulator On in Non Power Cut Modes and Off in Power Cut Modes

VBKUP_MODE4 - VBKUPxEN = 1, VBKUPxAUTO = 1

Backup Regulator On in Non Power Cut Modes and On in Power Cut Modes

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.20 PmicPwrctrlSetVBKUPRegulatorVoltage

Prototype `PMIC_STATUS PmicPwrctrlSetVBKUPRegulatorVoltage (MC13783_PWRCTRL_REG_VBKUP reg, UINT8 volt);`

This function is used to set the VBKUP regulator voltage

Parameters: `reg [in]`

the backup regulator to set

`volt [in]`

the voltage to set to backup regulator

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.21 PmicPwrctrlEnableWarmStart

Prototype `PMIC_STATUS PmicPwrctrlEnableWarmStart (void);`

This function is used to set the phone to transit from the ON state to the User Off state when either the USER_OFF pin is pulled high or the USER_OFF_SPI bit is set (after an 8ms delay in the Memwait state).

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.22 PmicPwrctrlDisableWarmStart

Prototype `PMIC_STATUS PmicPwrctrlDisableWarmStart (void);`

This function is used to disable the warm start and set the phone to transit from the ON state to the MEMHOLD ONLY state when either the USER_OFF pin is pulled high or the USER_OFF_SPI bit is set (after an 8ms delay in the Memwait state).

Parameters: None

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.23 PmicPwrctrlEnableRegenAssig

Prototype `PMIC_STATUS PmicPwrctrlEnableRegenAssig (t_regulator regu);`

This function enables the REGEN pin of selected voltage regulator. The REGEN function can be used in two ways. It can be used as a regulator enable pin like with SIMEN where the SPI programming is static and the REGEN pin is dynamic. It can also be used in a static fashion where REGEN is maintained high while the regulators get enabled and disabled dynamically via SPI. In that case REGEN functions as a master enable.

Parameters: `t_regulator regu`

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.24 PmicPwrctrlDisableRegenAssig

Prototype `PMIC_STATUS PmicPwrctrlDisableRegenAssig (t_regulator regu);`

This function Disable the REGEN pin of selected voltage regulator.

Parameters: `t_regulator regu`

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.2.25 PmicPwrctrlGetRegenAssig

Prototype `PMIC_STATUS PmicPwrctrlGetRegenAssig (t_regulator regu , UINT8* value);`

This function reads the REGEN pin value for said voltage regulator.

Parameters: `t_regulator regu` , value

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.3 PowerCutTimer Functions

The maximum duration of a power cut is determined by the power cut timer PCT[7:0]. By SPI this timer is set to a preset value. When a power cut occurs the timer will internally be decremented till it expires, meaning counted down to zero. The contents of PCT[7:0] does not reflect the actual counted down value but will keep the programmed value and therefore does not have to be reprogrammed after each power cut.

Using the following functions enable/disable/maximum duration of a power cut is determined

```
PMIC_STATUS PmicPwrctrlSetPowerCutTimer (UINT8 duration);
PMIC_STATUS PmicPwrctrlGetPowerCutTimer (UINT8* duration);
PMIC_STATUS PmicPwrctrlEnablePowerCut (void);
PMIC_STATUS PmicPwrctrlDisablePowerCut (void);
```

The following code example shows how to use the power control functions of PMIC module.

```
int MC13783_power_cut_conf(struct t_power_cut_conf *pc)
{
        if(!pc->pc_counter_en)
        {
           PmicPwrctrlDisablePowerCut();
        }
        else
           PmicPwrctrlEnablePowerCut();
        if(!pc->pc_auto_user_off)
        {
           PmicPwrctrlDisableUserOffModeWhenDelay();
        }
        else
           PmicPwrctrlEnableUserOffModeWhenDelay();
        if(!pc->pc_auto_user_off)
        {
           PmicPwrctrlDisableClk32kMCU();
        }
        else
           PmicPwrctrlEnableClk32kMCU();

        if(pc->pc_timer)
           PmicPwrctrlSetPowerCutTimer (pc->pc_timer);
        if(pc->pc_counter)
           PmicPwrctrlSetPowerCutCounter(pc->pc_counter);
        if(pc->pc_max_nb_pc)
           PmicPwrctrlSetPowerCutMaxCounter(pc->pc_max_nb_pc);
        if(pc->pc_ext_timer)
           PmicPwrctrlSetMemHoldTimer (pc->pc_ext_timer);
        if(pc->pc_ext_timer_inf)
           PmicPwrctrlSetMemHoldTimerAllOn();
        else
        PmicPwrctrlClearMemHoldTimerAllOn();
        return 0;
}
```

24.11.4 Memory Hold Operation functions

The Memory Hold circuit provides power to the memory during a power cut via VBKUP1. To avoid leakage from the VBKUP1 into circuitry connected to BP during a power cut, an external PMOS is to be placed between the memory supplies.

Following functions are used to set/get the duration of memory hold timer.

```
PMIC_STATUS PmicPwrctrlSetMemHoldTimer (UINT8 duration)
PMIC_STATUS PmicPwrctrlGetMemHoldTimer (UINT8* duration)
```

Following functions are used to set/clear the duration of the memory hold timer to infinity

```
PMIC_STATUS PmicPwrctrlSetMemHoldTimerAllOn (void)
PMIC_STATUS PmicPwrctrlClearMemHoldTimerAllOn (void)
```

The following code example shows how to use the power controller memory hold operation functions of PMIC module.

```
int MC13783_power_cut_get_conf(struct t_power_cut_conf *pc)
{
         UINT8 duration;
         UINT8 counter;
         UINT32 reg;
         unsigned char index;
         PmicPwrctrlGetPowerCutTimer (&duration);
         pc->pc_timer = duration;

         PmicPwrctrlGetPowerCutCounter (&counter);
         pc->pc_counter = counter;

         PmicPwrctrlGetPowerCutMaxCounter(&duration);
         pc->pc_max_nb_pc = duration;
         PmicPwrctrlGetMemHoldTimer (&duration);
         pc->pc_ext_timer = duration;
         index = 0x0E;MC13783_PWR_CTL1_ADDR
         PmicRegisterRead(index, &reg);
         if((reg &0x00100000))
            pc->pc_ext_timer_inf = 1;((reg &0x00100000) >> 20);
         else
            pc->pc_ext_timer_inf = 0;

         pc->pc_max_nb_pc = ((reg &0x0000F800) >> 11);

         PmicPwrctrlGetMemHoldTimer (&duration);
         pc->pc_ext_timer=duration;

         index = 0x0D;MC13783_PWR_CTL0_ADDR
         PmicRegisterRead(index, &reg);
         if((reg &0x00000002))
            pc->pc_counter_en = 1;
         else
            pc->pc_counter_en = 0;

         if((reg &0x00000008))
```

```
            pc->pc_auto_user_off =1;
         else
            pc->pc_auto_user_off =0;

         if((reg &0x00000020))
            pc->pc_user_off_32k_en=1;
         else
            pc->pc_user_off_32k_en=0;

return 0;
}
```

24.11.5 Power Cut Counter Functions

PwCtrl provides a method for disabling power cuts if this situation manifests itself. If PC_COUNT >= PC_MAX_COUNT, then the number of resets that have occurred since the power cut counter was last initialized exceeds the established limit, and power cuts will be disabled. PwCtrl counters can be disabled by setting PC_COUNT_EN=0. In this case, each power cut event will increment the power cut counter, but power cut coverage will not be disabled, even if PC_COUNT exceeds PC_MAX_COUNT.

Following functions are used to set/get the power cut counter values.

```
PMIC_STATUS PmicPwrctrlSetPowerCutCounter (UINT8 counter)
PMIC_STATUS PmicPwrctrlGetPowerCutCounter (UINT8* counter)
PMIC_STATUS PmicPwrctrlSetPowerCutMaxCounter (UINT8 counter)
PMIC_STATUS PmicPwrctrlGetPowerCutMaxCounter (UINT8* counter)
```

Following functions are used to enable/disable the duration of the power cut counters.

```
PMIC_STATUS PmicPwrctrlEnablePowerCut (void)
PMIC_STATUS PmicPwrctrlDisablePowerCut (void)
```

The following code example shows how to use the power controller power cut counter functions of PMIC module. Some the functions are used in the above examples.

```
int MC13783_power_cut_get_conf(struct t_power_cut_conf *pc)
{
         UINT8 duration;
         UINT8 counter;
         UINT32 reg;
         unsigned char index;
         PmicPwrctrlGetPowerCutTimer (&duration);
         pc->pc_timer = duration;

         PmicPwrctrlGetPowerCutCounter (&counter);
         pc->pc_counter = counter;

         PmicPwrctrlGetPowerCutMaxCounter(&duration);
         pc->pc_max_nb_pc = duration;
         PmicPwrctrlGetMemHoldTimer (&duration);
         pc->pc_ext_timer = duration;
         index = 0x0E;MC13783_PWR_CTL1_ADDR
         PmicRegisterRead(index, &reg);
         if((reg &0x00100000))
```

```
            pc->pc_ext_timer_inf = 1;((reg &0x00100000) >> 20);
         else
            pc->pc_ext_timer_inf = 0;

         pc->pc_max_nb_pc = ((reg &0x0000F800) >> 11);

         PmicPwrctrlGetMemHoldTimer (&duration);
         pc->pc_ext_timer=duration;

         index = 0x0D;MC13783_PWR_CTL0_ADDR
         PmicRegisterRead(index, &reg);
         if((reg &0x00000002))
            pc->pc_counter_en = 1;
         else
            pc->pc_counter_en = 0;

         if((reg &0x00000008))
            pc->pc_auto_user_off =1;
         else
            pc->pc_auto_user_off =0;

         if((reg &0x00000020))
            pc->pc_user_off_32k_en=1;
         else
            pc->pc_user_off_32k_en=0;
         return 0;
}
```

24.11.6 Power Management

There is no additional power management implementation done specifically for Atlas Power Control other than the implementation described in section Power Management of this document.

24.11.7 Voltage Regulator

The ARM11/ARM9 processor cores and memories are supposed to be supplied by the switchers. All other building blocks are supplied either directly from the battery or via a linear regulator.

For convenience these regulators are labeled to indicate their intended purpose. This concerns VRF1 and VRF2 for the transceiver transmit and receive supplies, VRFREF, VRFBG and VRFCP as the transceiver references, VRFDIG, VDIG and VGEN for the different digital sections of the platform, VIOHI, VIOLO for the different interfaces, VCAM for the camera module, VSIM1 for the SIM card, VESIM1 for the eSIM card, VMMC1 and VMCC2 for dual multimedia card support or peripheral supply such as Bluetooth PA.

24.11.8 Data Structures

```
// switch mode regulator
typedef enum _MC13783_REGULATOR_SREG{
    SW1A = 0,
    SW1B,
```

```
    SW2A,
    SW2B,
    SW3,
} MC13783_REGULATOR_SREG;
typedef MC13783_REGULATOR_SREG PMIC_REGULATOR_SREG;

typedef UINT8 PMIC_REGULATOR_SREG_VOLTAGE;


/***********************************
 * Switch regulator voltage settings type:
 *
 * SW_VOLTAGE_NORMAL
 * SW_VOLTAGE_DVS
 * SW_VOLTAGE_STBY
 *
 ***********************************/
typedef enum _RR_REGULATOR_SREG_VOLTAGE_TYPE{
    SW_VOLTAGE_NORMAL=0,
    SW_VOLTAGE_DVS,
    SW_VOLTAGE_STBY,
 } RR_REGULATOR_SREG_VOLTAGE_TYPE;
typedef RR_REGULATOR_SREG_VOLTAGE_TYPE PMIC_REGULATOR_SREG_VOLTAGE_TYPE;


// standby input state H/L
typedef enum _MC13783_REGULATOR_SREG_STBY{
    LOW = 0,
    HIGH,
}MC13783_REGULATOR_SREG_STBY;
typedef MC13783_REGULATOR_SREG_STBY PMIC_REGULATOR_SREG_STBY;


/***********************************************
// switch regulator modes:
//              1. OFF
//              2. PWM mode and no Pulse Skipping
//              3. PWM mode and pulse Skipping Allowed
//              4. Low Power PFM mode
***********************************************/
typedef enum _MC13783_REGULATOR_SREG_MODE{
    SW_MODE_OFF,
    SW_MODE_PWM,
    SW_MODE_PULSESKIP,
    SW_MODE_PFM,
}MC13783_REGULATOR_SREG_MODE;
typedef MC13783_REGULATOR_SREG_MODE PMIC_REGULATOR_SREG_MODE;

// linear voltage regulator
typedef enum _MC13783_REGULATOR_VREG{
    VIOHI = 0,
    VIOLO,
```

```
    VDIG,
    VGEN,
    VRFDIG,
    VRFREF,
    VRFCP,
    VSIM,
    VESIM,
    VCAM,
    V_VIB,
    VRF1,
    VRF2,
    VMMC1,
    VMMC2,
} MC13783_REGULATOR_VREG;
typedef MC13783_REGULATOR_VREG PMIC_REGULATOR_VREG;


/*************************************************************************
// LOW_POWER
// VxMODE=1, Set Low Power no matter of VxSTBY and STANDBY pin
//
// LOW_POWER_CTL_BY_PIN
// VxMODE=0, VxSTBY=1, Low Power Mode is controlled by STANDBY pin
//
// LOW_POWER_DISABLED
// VxMODE=0, VxSTBY=0, Low Power Mode is disabled
*************************************************************************/
typedef enum _MC13783_REGULATOR_VREG_POWER_MODE{
    LOW_POWER_DISABLED = 0,
    LOW_POWER,
    LOW_POWER_CTRL_BY_PIN,
} MC13783_REGULATOR_VREG_POWER_MODE;
typedef MC13783_REGULATOR_VREG_POWER_MODE PMIC_REGULATOR_VREG_POWER_MODE;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VIOHI{
    VIOHI_2_775 = 0,   //output  2.775V,
 } MC13783_REGULATOR_VREG_VOLTAGE_VIOHI;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VIOLO{
    VIOLO_1_20V = 0,   //output  1.20V,
    VIOLO_1_30V,        //output   1.30V,
    VIOLO_1_50V,        //output   1.50V,
    VIOLO_1_80V,        //output   1.80V,
} MC13783_REGULATOR_VREG_VOLTAGE_VIOLO;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VRFDIG{
    VRFDIG_1_20V = 0,   //output  1.20V,
    VRFDIG_1_50V,        //output   1.50V,
    VRFDIG_1_80V,        //output   1.80V,
    VRFDIG_1_875V,        //output   1.875V,
} MC13783_REGULATOR_VREG_VOLTAGE_VRFDIG;
```

```
typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VDIG{
    VDIG_1_20V = 0,   //output  1.20V,
    VDIG_1_30V,        //output   1.30V,
    VDIG_1_50V,        //output   1.50V,
    VDIG_1_80V,        //output   1.80V,
} MC13783_REGULATOR_VREG_VOLTAGE_VDIG;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VGEN{
    VGEN_1_20V = 0,   //output  1.20V,
    VGEN_1_30V,        //output   1.30V,
    VGEN_1_50V,        //output   1.50V,
    VGEN_1_80V,        //output   1.80V,
} MC13783_REGULATOR_VREG_VOLTAGE_VGEN;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VRF{
    VRF2_1_875V = 0,   //output  1.875V,
    VRF2_2_475V,        //output   2.475V,
    VRF2_2_700V,        //output   2.700V,
    VRF2_2_775V,        //output   2.775V,
} MC13783_REGULATOR_VREG_VOLTAGE_VRF;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VRFCP{
    VRFCP_2_700V = 0,   //output  2.700V,
    VRFCP_2_775V,        //output   2.775V,
} MC13783_REGULATOR_VREG_VOLTAGE_VRFCP;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VRFREF{
    VRFREF_2_475V = 0,   //output  2.475V,
    VRFREF_2_600V,        //output   2.600V,
    VRFREF_2_700V,        //output   2.700V,
    VRFREF_2_775V,        //output   2.775V,
} MC13783_REGULATOR_VREG_VOLTAGE_VRFREF;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_CAM{
                          //        1st silicon,  2nd silicon
    VCAM_1 = 0,   //output  1.50V,          1.5V.
    VCAM_2,        //output   1.80V,          1.80V
    VCAM_3,        //output   2.50V,          2.50V
    VCAM_4,        //output   2.80V,          2.55V
    VCAM_5,        //output    -                2.60V
    VCAM_6,        //output    -                2.80V
    VCAM_7,        //output    -                3.00V
    VCAM_8,        //output    -                TBD
} MC13783_REGULATOR_VREG_VOLTAGE_CAM;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_SIM{
    VSIM_1_8V = 0,   //output = 1.80V
    VSIM_2_9V,   //output = 2.90V
} MC13783_REGULATOR_VREG_VOLTAGE_SIM;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_ESIM{
    VESIM_1_8V = 0,   //output = 1.80V
```

```
    VESIM_2_9V,   //output = 2.90V
} MC13783_REGULATOR_VREG_VOLTAGE_ESIM;


typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_MMC{
                    //        1st silicon,  2nd silicon
    VMMC_1,  //output    1.60V,           1.60V
    VMMC_2,  //output    1.80V,           1.80V
    VMMC_3,  //output    2.00V,           2.00V
    VMMC_4, //output     2.20V,           2.60V
    VMMC_5,  //output    2.40V,           2.70V
    VMMC_6,  //output    2.60V,           2.80V
    VMMC_7,  //output    2.80V,           2.90V
    VMMC_8,  //output    2.90V,           3.00V
} MC13783_REGULATOR_VREG_VOLTAGE_MMC;

typedef enum _MC13783_REGULATOR_VREG_VOLTAGE_VIB{
    V_VIB_1_3V = 0,   //output = 1.30V
    V_VIB_1_8V,   //output = 1.80V
    V_VIB_2_0V,   //output = 2.0V
    V_VIB_3_0V,   //output = 3.0V
} MC13783_REGULATOR_VREG_VOLTAGE_VIB;

typedef union {
    MC13783_REGULATOR_VREG_VOLTAGE_VIOHI viohi;
    MC13783_REGULATOR_VREG_VOLTAGE_VIOLO violo;
    MC13783_REGULATOR_VREG_VOLTAGE_VRFDIG vrfdig;
    MC13783_REGULATOR_VREG_VOLTAGE_VDIG vdig;
    MC13783_REGULATOR_VREG_VOLTAGE_VGEN vgen;
    MC13783_REGULATOR_VREG_VOLTAGE_VRF vrf;
    MC13783_REGULATOR_VREG_VOLTAGE_VRFCP vrfcp;
    MC13783_REGULATOR_VREG_VOLTAGE_VRFREF vrfref;
    MC13783_REGULATOR_VREG_VOLTAGE_CAM vcam;
    MC13783_REGULATOR_VREG_VOLTAGE_SIM vsim;
    MC13783_REGULATOR_VREG_VOLTAGE_ESIM vesim;
    MC13783_REGULATOR_VREG_VOLTAGE_MMC vmmc;
    MC13783_REGULATOR_VREG_VOLTAGE_VIB v_vib;
} MC13783_REGULATOR_VREG_VOLTAGE;
typedef MC13783_REGULATOR_VREG_VOLTAGE PMIC_REGULATOR_VREG_VOLTAGE;

typedef enum _MC13783_REGULATOR_ENABLE{
    DISABLE = 0,
    ENABLE = 1,
} MC13783_REGULATOR_ENABLE;
```

24.11.9 Switch mode regulator API's

24.11.9.1 PmicSwitchModeRegulatorOn

Prototype *PMIC_STATUS PmicSwitchModeRegulatorOn (PMIC_REGULATOR_SREG regulator);*

Parameters: *regulator [in]*

Which switch mode regulator to turn on

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failureThis function is used to turn on the switch mode regulator.

24.11.9.2 PmicSwitchModeRegulatorOff

Prototype *PMIC_STATUS PmicSwitchModeRegulatorOff (PMIC_REGULATOR_SREG regulator);*

This function is used to turn off the switch regulator

Parameters: *regulator [in]*

Which switch mode regulator to turn off

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.3 PmicSwitchModeRegulatorSetVoltageLevel

Prototype *PMIC_STATUS PmicSwitchModeRegulatorSetVoltageLevel (PMIC_REGULATOR_SREG regulator,*
PMIC_REGULATOR_SREG_VOLTAGE_TYPE voltageType,
PMIC_REGULATOR_SREG_VOLTAGE voltage);

This function is to set the voltage level for the switch regulator.

Parameters: *regulator [in]*

The regulator to be set

voltageType [in]
SW_VOLTAGE_NORMAL/SW_VOLTAGE_LVS/SW_VOLTAGE_STBY

SW1 offers support for Dynamic Voltage-Frequency scaling. If this feature is activated, then assertion of the STANDBY input will automatically configure SW1 to output the voltage defined by the 3-bit field SW1X_STBY. If STANDBY=LOW, then assertion of the LVS input will automatically configure SW1 to output the voltage defined by the 3-bit field SW1X_LVS. These alternative bit fields would normally be programmed to a voltage lower than that encoded in the SW1X bit field. When STANDBY and LVS are both de-asserted, the output voltage will revert the that encoded by the SW1X field.

SW2 offers limited support for Dynamic Voltage-Frequency scaling. If this feature is activated, then assertion of the STANDBY input will automatically configure SW2 to output the voltage defined by the 3-bit field SW2_STBY.

If STANDBY=LOW, then assertion of the LVS2 input will automatically configure SW2 to output the voltage defined by the 3-bit SW2X_LVS field. When STANDBY and LVS2 are both de-asserted, the output voltage will revert to that encoded by the SW2X 3-bit field.

voltage [in]

The voltage to be set, it depends on different regulator.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.4 PmicSwitchModeRegulatorGetVoltageLevel

Prototype

```
PMIC_STATUS PmicSwitchModeRegulatorGetVoltageLevel (PMIC_REGULATOR_SREG
regulator, PMIC_REGULATOR_SREG_VOLTAGE_TYPE voltageType,
PMIC_REGULATOR_SREG_VOLTAGE* voltage);
```

This function is to get the voltage settings.

Parameters: *regulator [in]*

The regulator to get voltage from

voltageType [in]

SW_VOLTAGE_NORMAL/SW_VOLTAGE_LVS/SW_VOLTAGE_STBY

voltage [out]

the pointer to get the value

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.5 PmicSwitchModeRegulatorSetMode

Prototype

```
PMIC_STATUS PmicSwitchModeRegulatorSetMode (PMIC_REGULATOR_SREG
regulator, PMIC_REGULATOR_SREG_MODE mode);
```

This function is to set the switch mode regulator into synchronous rectifier mode or pulse skipping mode.The synchronous rectifier can be disabled (and pulse-skipping enabled) to improve low current efficiency. Software should disable synchronous rectifier / enable the pulse skipping for average loads less than approximately 30 mA, depending on the quiescent current penalty due to synchronous mode.

Parameters: *regulator [in]*

The regulator to be set

mode [in]

Synchronous rectifier mode or pulse skipping mode.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.6 PmicSwitchModeRegulatorGetMode

Prototype

```
PMIC_STATUS PmicSwitchModeRegulatorGetMode (PMIC_REGULATOR_SREG
regulator, PMIC_REGULATOR_SREG_MODE* mode);
```

This function get the current setting of regulator mode

Parameters: *regulator [in]*

The regulator to get voltage value from

mode [out]

Synchronous rectifier mode or pulse skipping mode.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.7 PmicSwitchModeRegulatorEnableSTBYDVFS

Prototype *PMIC_STATUS PmicSwitchModeRegulatorEnableSTBYDVFS (PMIC_REGULATOR_SREG regulator);*

This function is used to enable the standby or Dynamic Voltage-Frequency scaling.

Parameters: *regulator [in]*

The regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.8 PmicSwitchModeRegulatorDisableSTBYDVFS

Prototype *PMIC_STATUS PmicSwitchModeRegulatorDisableSTBYDVFS (PMIC_REGULATOR_SREG regulator);*

This function is used to disable the standby or Dynamic Voltage-Frequency scaling.

Parameters: *regulator [in]*

The regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.9.9 PmicSwitchModeRegulatorSetDVSSpeed

Prototype *PMIC_STATUS PmicSwitchModeRegulatorSetDVSSpeed (PMIC_REGULATOR_SREG regulator, UINT8 dvsspeed);*

This function is to set the DVS speed the regulator.

Parameters: *regulator [in]*

The regulator to be set

dvsspeed [in]

The speed settings for DVS

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: This function is only applicable to MC13783, it is a stub function here.

24.11.9.10 PmicSwitchModeRegulatorEnablePanicMode

Prototype `PMIC_STATUS PmicSwitchModeRegulatorEnablePanicMode(PMIC_REGULATOR_SREG regulator);`

This function is used to enable the panic mode.

Parameters: `regulator [in]`

The regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: This is a stub function here.

24.11.9.11 PmicSwitchModeRegulatorDisablePanicMode

Prototype `PMIC_STATUS PmicSwitchModeRegulatorDisablePanicMode(PMIC_REGULATOR_SREG regulator);`

This function is used to disable the panic mode.

Parameters: `regulator [in]`

the regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: This is a stub function here.

24.11.9.12 PmicSwitchModeRegulatorEnableSoftStart

Prototype `PMIC_STATUS PmicSwitchModeRegulatorEnableSoftStart(PMIC_REGULATOR_SREG regulator);`

This function is used to enable soft start.

Parameters: `regulator [in]`

The regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: This is a stub function here.

24.11.9.13 PmicSwitchModeRegulatorDisableSoftStart

Prototype `PMIC_STATUS PmicSwitchModeRegulatorDisableSoftStart(PMIC_REGULATOR_SREG regulator);`

This function is used to disable soft start.

Parameters: *regulator [in]*

The regulator to be set

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: This is a stub function here.

24.11.10 Linear Voltage Regulator API's

24.11.10.1 PmicVoltageRegulatorOn

Prototype *PMIC_STATUS PmicVoltageRegulatorOn (PMIC_REGULATOR_VREG regulator);*

This function is used to turn on the voltage regulator

Parameters: *regulator [in]*

Which voltage regulator to turn on

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: MMC does not have on/off, just directly set the voltage level.

0V=OFF will return PMIC_INVALID_PARAMETER.

24.11.10.2 PmicVoltageRegulatorOff

Prototype *PMIC_STATUS PmicVoltageRegulatorOff (PMIC_REGULATOR_VREG regulator);*

This function is used to turn off the regulator

Parameters: *regulator [in]*

Which voltage regulator to turn off

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

Remarks: MMC don't have on/off, just directly set the voltage level. 0V=OFF will return PMIC_INVALID_PARAMETER.

24.11.10.3 PmicVoltageRegulatorSetVoltageLevel

Prototype *PMIC_STATUS PmicVoltageRegulatorSetVoltageLevel (PMIC_REGULATOR_VREG regulator, PMIC_REGULATOR_VREG_VOLTAGE voltage);*

This function is used to set voltage level for the voltage regulator.

Parameters: *regulator [in]*

Which switch mode regulator to be set

voltage [in]

The voltage value to be set to the register

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.10.4 PmicVoltageRegulatorGetVoltageLevel

Prototype `PMIC_STATUS PmicVoltageRegulatorGetVoltageLevel (PMIC_REGULATOR_VREG regulator, PMIC_REGULATOR_VREG_VOLTAGE* voltage);`

This function is to get the current voltage settings of the regulator.

Parameters: `regulator [in]`

Which switch mode regulator to get the value from

`voltage [out]`

the pointer to storage the return value

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.10.5 PmicVoltageRegulatorSetPowerMode

Prototype `PMIC_STATUS PmicVoltageRegulatorSetPowerMode (PMIC_REGULATOR_VREG regulator, PMIC_REGULATOR_VREG_POWER_MODE powerMode);`

This function is used to set low power mode for the regulator and whether to enter low power mode during STANDBY assertion or not.

Parameters: `regulator [in]`

Which switch mode regulator to be set

`powerMode[in]`

LOW_POWER

VxMODE=1, Set Low Power no matter of VxSTBY and STANDBY pin

LOW_POWER_CTL_BY_PIN

VxMODE=0, VxSTBY=1, Low Power Mode is controlled by STANDBY pin
LOW_POWER_DISABLED

VxMODE=0, VxSTBY=0, Low Power Mode is disabled.

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure

24.11.10.6 PmicVoltageRegulatorGetPowerMode

Prototype `PMIC_STATUS PmicVoltageRegulatorGetPowerMode (PMIC_REGULATOR_VREG regulator, PMIC_REGULATOR_VREG_POWER_MODE* powerMode);`

This function is to get the current power mode for the regulator

Parameters: `regulator [in]`

Which switch mode regulator to get the value from

powerMode [out]

Pointer to storage the powerMode get from the register

Returns: *status*

PMIC_SUCCESS for success and PMIC_ERROR for failure.

24.11.11 Power Management

There is no additional power management implementation done specifically for Atlas Voltage Regulator other than the implementation described in section of this document.

24.11.12 Battery Charger

24.11.13 Data Structures

```
typedef enum {
        BATT_MAIN_CHGR = 0,         // Main battery charger
        BATT_CELL_CHGR,             // CoinCell battery charger
        BATT_TRCKLE_CHGR            // Trickle charger
} BATT_CHARGER;

typedef enum {
    DUAL_PATH = 0,
    SINGLE_PATH,
    SERIAL_PATH,
    DUAL_INPUT_SINGLE_PATH,
    DUAL_INPUT_SERIAL_PATH,
    INVALID_CHARGER_MODE
}CHARGER_MODE;

typedef enum {
    LOW = 0,  //GND
    OPEN,     //HI Z
    HIGH      //VMC13783
}CHARGERMODE_PIN;
```

24.11.14 Battery Charger API (Compatible with SC55112 API)

24.11.14.1 PmicBatterEnableCharger

This function is used to start charging a battery. For different charger, different voltage and current range are supported.

The main battery charger supports a settable voltage and current. The coincell supports only a settable voltage. The trickel charger only a settable current.

Prototype *PMIC_STATUS PmicBatterEnableCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);*

Parameters: *chgr [in]*

Charger as defined in BATT_CHARGER

c_voltage [in]

Charging voltage. (main and coincell)

c_current [in]

Charging current. (main and trickle)

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.2 PmicBatterDisableCharger

This function turns off the selected charger. This is done by setting the curren level to zero for the main and trickele chargers. The coincell charger is disabled.

Prototype *PMIC_STATUS PmicBatterDisableCharger(BATT_CHARGER chgr)*

Parameters: *chgr [in]*

Charger as defined in BATT_CHARGER.

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.3 PmicBatterSetCharger

This function is used to change the charger setting.

Prototype *PMIC_STATUS PmicBatterSetCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);*

Parameters: *chgr [in]*

Charger as defined in BATT_CHARGER

c_voltage [in]

Charging voltage. (main and coincell)

c_current [in]

Charging current. (main and trickle)

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.4 PmicBatterGetChargerSetting

This function is used to retrieve what the charger setting are for the selected charger not what is measured

Prototype *PMIC_STATUS PmicBatterGetChargerSetting(BATT_CHARGER chgr, UINT8* c_voltage, UINT8* c_current);*

Parameters: *chgr [in]*

Charger as defined in BATT_CHARGER

**c_voltage [out]*

A pointer to what the charging voltage is set to. (main and coincell)

c_current [out]

A pointer to what the charging current is set to. (main and trickle)

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.5 PmicBatterGetChargeCurrent

This function is retrieves the main charger current. This value is obtained by reading a voltage between CHRGISNSP – CHRGISNSN. This corresponds to ADC channel 4.

Prototype *PMIC_STATUS PmicBatterGetChargeCurrent(UINT16* c_current);*

Parameters: **c_current [out]*

A pointer to what the measured charger current

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.6 PmicBatterEnableEol

This function enables End-of-Life comparator.

Prototype *PMIC_STATUS PmicBatterEnableEol(void);*

Parameters: None

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.7 PmicBatterDisableEol

This function disables End-of-Life comparator.

Prototype *PMIC_STATUS PmicBatterDisableEol (void);*

Parameters: None

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.8 PmicBatterLedControl

This function controls charge LED.

Prototype *PMIC_STATUS PmicBatterLedControl(BOOL on);*

Parameters: *on [in]*

If on is true, LED will be turned on, or otherwise the LED will be turned off.

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.9 PmicBatterSetReverseSupply

This function sets reverse supply mode.

Prototype *PMIC_STATUS PmicBatterSetReverseSupply(BOOL enable);*

Parameters: *enable [i]*

If enable is true, reverse supply mode is enable or otherwise the reverse supply mode is disabled.

Returns: This function returns PMIC_SUCCESS if successful.

24.11.14.10 PmicBatterSetUnregulated

This function sets limited charging mode on main battery charger. If this mode is selected the current is no longer controlled and it is only limited by what the charger can supply.

Prototype `PMIC_STATUS PmicBatterSetUnregulated(BOOL enable);`

Parameters: `enable [in]`

If enable is true, unregulated charging mode is enabled otherwise it is disabled.

Returns: This function returns PMIC_SUCCESS if successful.

24.11.15 Battery Charger API (MC13783 Native For Compatibility with SC55112)

These functions are just there for compatibility with the SC55112 API. This is an effort to maintain one PMIC API, regardless of which PMIC is being used. These are implemented as a stubs returning a PMIC_STATUS of PMIC_SUCCESS.

24.11.15.1 PmicBatteryEnableAdChannel5

This function enables use of AD channel 5 to read charge current on the SC55112 PMIC.

Prototype `PMIC_STATUS PmicBatteryEnableAdChannel5();`

Parameters: None.

Returns: PMIC_SUCCESS

24.11.15.2 PmicBatteryDisableAdChannel5

This function disables use of AD channel 5 to read charge current on the SC55112 PMIC.

Prototype `PMIC_STATUS PmicBatteryDisableAdChannel5();`

Parameters: None.

Returns: PMIC_SUCCESS;

24.11.15.3 PmicBatterySetCoincellCurrentlimit

This function limits the output current level of coincell charger on the SC55112 PMIC.

Prototype `PMIC_STATUS PmicBatterySetCoincellCurrentlimit (UINT8 coincellcurrentlevel);`

Parameters: `coincellcurrentlevel [IN]`

coincell current level

Returns: PMIC_SUCCESS;

24.11.15.4 PmicBatteryGetCoincellCurrentlimit

This function returns the output current limit of coincell charger on the SC55112 PMIC.

Prototype *PMIC_STATUS PmicBatteryGetCoincellCurrentlimit (UINT8* coincellcurrentlevel);*

Parameters: *coincellcurrentlevel [OUT]*

Pointer to coincell current level

Returns: PMIC_SUCCESS;

24.11.15.5 PmicBatterySetEolTrip

This function sets the end-of-life threshold on the SC55112 PMIC.

Prototype *PMIC_STATUS PmicBatterySetEolTrip (UINT8 eoltriplevel);*

Parameters: *eoltriplevel [IN]*

eol trip level

Returns: PMIC_SUCCESS;

24.11.15.6 PmicBatteryGetEolTrip

This function returns the end-of-life threshold on the SC55112 PMIC.

Prototype *PMIC_STATUS PmicBatteryGetEolTrip (UINT8* eoltriplevel);*

Parameters: *eoltriplevel [OUT]*

pointer to eol trip level

Returns: PMIC_SUCCESS;

24.11.16 Battery Charger API (MC13783 Native)

24.11.16.1 PmicBatterySetChargeVoltage

This function programs the output voltage of charge regulator.

Prototype *PMIC_STATUS PmicBatterySetChargeVoltage(UINT8 chargevoltagelevel);*

Parameters: *chargevoltagelevel [IN]*

voltage level

level 0 = 4.05V

1 = 4.10V

2 = 4.15V

3 = 4.20V

4 = 4.25V

5 = 4.30V

6 = 3.80V ... lowest setting

7 = 4.50V

Returns: PMIC_STATUS

24.11.16.2 PmicBatteryGetChargeVoltage

This function returns the output voltage of charge regulator.

Prototype *PMIC_STATUS PmicBatteryGetChargeVoltage(UINT8* chargevoltagelevel);*

Parameters: *chargevoltagelevel [OUT]*

pointer to voltage level

Returns: PMIC_STATUS

24.11.16.3 PmicBatterySetChargeCurrent

This function programs the charge current limit level to the main battery.

Prototype *PMIC_STATUS PmicBatterySetChargeCurrent (UINT8 chargecurrentlevel);*

Parameters: *chargecurrentlevel [IN]*

current level

level 0 = 0 mA (max value)

1 = 100 mA (max value)

... (in increment of 100 mA)

13 = 1300 mA (max value)

14 = 1800 mA (max value)

15 = disables the current limit

Returns: PMIC_STATUS

24.11.16.4 PmicBatteryGetChargeCurrent

This function returns the charge current setting of the main battery.

Prototype *PMIC_STATUS PmicBatteryGetChargeCurrent (UINT8* chargecurrentlevel);*

Parameters: *chargecurrentlevel [OUT]*

pointer to current level

Returns: PMIC_STATUS

24.11.16.5 PmicBatterySetTrickleCurrent

This function programs the current of the trickle charger.

Prototype *PMIC_STATUS PmicBatterySetTrickleCurrent(UINT8 tricklecurrentlevel);*

Parameters: *tricklecurrentlevel [IN]*

trickle current level

level 0 = 0 mA

1 = 12 mA
... (in addition of 12 mA per level)
6 = 72 mA
7 = 84 mA

Returns: PMIC_STATUS

24.11.16.6 PmicBatteryGetTrickleCurrent

This function returns the current of the trickle charger.

Prototype `PMIC_STATUS PmicBatteryGetTrickleCurrent (UINT8* tricklecurrentlevel);`

Parameters: `tricklecurrentlevel [OUT]`

pointer to trickle current level

Returns: PMIC_STATUS

24.11.16.7 PmicBatteryFETControl

This function programs the control mode and setting of BPFET and FETOVRD BATTFET and BPFET to be controlled by FETCTRL bit or hardware.

Prototype `PMIC_STATUS PmicBatteryFETControl(UINT8 fetcontrol);`

Parameters: `fetcontrol [IN]`

BPFET and FETOVRD control mode and setting
input = 0 (BATTFET and BPFET outputs are controlled by hardware)
= 1 (BATTFET and BPFET outputs are controlled by hardware)
= 2 (BATTFET low and BATTFET high, controlled by FETCTRL)
= 3 (BATTFET high and BATTFET low, controlled by FETCTRL)

Returns: PMIC_STATUS

24.11.16.8 PmicBatteryReverseDisable

This function disables the reverse mode.

Prototype `PMIC_STATUS PmicBatteryReverseDisable();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.9 PmicBatteryReverseEnable

This function enables the reverse mode.

Prototype `PMIC_STATUS PmicBatteryReverseEnable();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.10 PmicBatterySetOvervoltageThreshold

This function programs the overvoltage threshold value.

Prototype `PMIC_STATUS PmicBatterySetOvervoltageThreshold(UINT8 ovthresholdlevel);`

Parameters: `ovthresholdlevel [IN]`

overvoltage threshold level

High to low, Low to High (5.35V)

Returns: PMIC_STATUS

24.11.16.11 PmicBatteryGetOvervoltageThreshold

This function returns the overvoltage threshold value.

Prototype `PMIC_STATUS PmicBatteryGetOvervoltageThreshold (UINT8* ovthresholdlevel);`

Parameters: `ovthresholdlevel [OUT]`

pointer to overvoltage threshold level

Returns: PMIC_STATUS

24.11.16.12 PmicBatteryUnregulatedChargeDisable

This function disables the unregulated charge path. The voltage and current limits will be controlled by the charge path regulator.

Prototype `PMIC_STATUS PmicBatteryUnregulatedChargeDisable();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.13 PmicBatteryUnregulatedChargeEnable

This function enables the unregulated charge path. The settings of the charge path regulator (voltage and current limits) will be overruled.

Prototype `PMIC_STATUS PmicBatteryUnregulatedChargeEnable();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.14 PmicBatteryChargeLedDisable

This function disables the charging LED.

Prototype `PMIC_STATUS PmicBatteryChargeLedDisable();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.15 PmicBatteryChargeLedEnable

This function enables the charging LED.

Prototype *PMIC_STATUS PmicBatteryChargeLedEnable();*

Parameters: None

Returns: PMIC_STATUS

24.11.16.16 PmicBatteryEnablePulldown

This function enables the 5k pull-down resistor used in the dual path charging.

Prototype *PMIC_STATUS PmicBatteryEnablePulldown();*

Parameters: None.

Returns: PMIC_STATUS.

24.11.16.17 PmicBatteryDisablePulldown

This function disables the 5k pull-down resistor used in the dual path charging.

Prototype *PMIC_STATUS PmicBatteryDisablePulldown();*

Parameters: None.

Returns: PMIC_STATUS.

24.11.16.18 PmicBatteryEnableCoincellCharger

This function enables the coincell charger.

Prototype *PMIC_STATUS PmicBatteryEnableCoincellCharger();*

Parameters: None

Returns: PMIC_STATUS

24.11.16.19 PmicBatteryDisableCoincellCharger

This function disables the coincell charger.

Prototype *PMIC_STATUS PmicBatteryDisableCoincellCharger();*

Parameters: None

Returns: PMIC_STATUS

24.11.16.20 PmicBatterySetCoincellVoltage

This function programs the output voltage level of coincell charger.

Prototype *PMIC_STATUS PmicBatterySetCoincellVoltage (UINT8 coincellvoltagelevel);*

Parameters: *votlagelevel [IN]*

voltage level

level 0 = 2.7V
1 = 2.8V
2 = 2.9V
... (in 100mV increment)
6 = 3.3V

Returns: PMIC_STATUS

24.11.16.21 PmicBatteryGetCoincellVoltage

This function returns the output voltage level of coincell charger.

Prototype `PMIC_STATUS PmicBatteryGetCoincellVoltage (UINT8* coincellvoltagelevel);`

Parameters: `voltagelevel [OUT]`

pointer to voltage level

Returns: PMIC_STATUS

24.11.16.22 PmicBatteryEnableEolComparator

This function enables the end-of-life function instead of the LOBAT.

Prototype `PMIC_STATUS PmicBatteryEnableEolComparator();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.23 PmicBatteryDisableEolComparator

This function disables the end-of-life comparator function.

Prototype `PMIC_STATUS PmicBatteryDisableEolComparator();`

Parameters: None

Returns: PMIC_STATUS

24.11.16.24 PmicBatteryGetChargerMode

This function returns the charger mode (ie. Dual Path, Single Path, Serial Path, Dual Input Single Path and the Dual Input Serial Path).

Prototype `PMIC_STATUS PmicBatteryGetChargerMode(CHARGER_MODE *mode);`

Parameters: `mode [OUT]`

pointer to charger mode

Returns: PMIC_STATUS

24.11.17 Power Management

There is no additional power management implementation done specifically for Atlas Battery other than the implementation described in section Power Management of this document.

24.12 A/D Converter and Touch

The ADC is a 16 channel, 10-bit converter with a state machine to control the various models of operation. Read and write access to the A/D converter is accomplished through the SPI bus.

MC13783 A/D Channel Definition and Scanning Table		
AD_SEL	ADA[2:0]	Signal Read
0	000	BATT
0	001	BATTISNS
0	010	BPSNS
0	011	CHRGRAW
0	100	CHRGISNS
0	101	ADIN5/PTHEN
0	110	ADIN6/LICELL
0	111	ADIN7/DTHEN
1	000	ADIN8
1	001	ADIN9
1	010	ADIN10
1	011	ADIN11
1	100	TSX1
1	101	TSX2
1	110	TSY1
1	111	TSY2

MC13783 has a touch screen interrupt. This interrupt occurs when a pen-down event is detected. The Windows CE touch driver should handle these interrupt event. Please refer to Section 24.9, “Interrupt Handling for a description of interrupt handling.

24.12.1 Data Types

```
typedef enum _PMIC_ADC_CONVERTOR_MODE
{
      ADC_8CHAN_1X = 0,    // RAND = 0, 8 channels
      ADC_1CHAN_8X         // RAND = 1, reads 8 sequential values
} PMIC_ADC_CONVERTOR_MODE;
Touch Modes
typedef enum _MC13783_TOUCH_MODE {
      TM_INACTIVE = 0,
```

```
      TM_INTRUPT,
      TM_RESISTIVE,
      TM_POSITION,
} MC13783_TOUCH_MODE;
```

24.12.2 Functions

24.12.2.1 PmicADCGetSingleChannelOneSample

This function gets one channel one sample.

Prototype *PMIC_STATUS PmicADCGetSingleChannelOneSample(UINT16 channel, UINT16* pResult);*

Parameters: *channel [in]*

A selected channel.

pResult [out]

Pointer to the sampled value.

Returns: PMIC_STATUS

24.12.2.2 PmicADCGetSingleChannelEightSamples

This function gets one channel eight samples.

Prototype *PMIC_STATUS PmicADCGetSingleChannelEightSamples(UINT16 channel, UINT16* pResult);*

Parameters: *channel [in]*

A selected channel.

pResult [out]

Pointer to the sampled values (up to 8 sampled values).

Returns: PMIC_STATUS

24.12.2.3 PmicADCGetMultipleChannelsSamples

This function gets sample for multiple channels.

Prototype *PMIC_STATUS PmicADCGetMultipleChannelsSamples(UINT16 channels, UINT16* pResult);*

Parameters: *channels [in]*

Selected channels (up to 16 channels).

pResult [out]

Pointer to the sampled values (up to 16 sampled values).

Returns: PMIC_STATUS

24.12.2.4 PmicADCTouchRead

This function reads a touch screen sample.

Prototype `PMIC_STATUS PmicADCTouchRead(UINT16* x, UINT16* y);`

Parameters: `x [out]`

X-coordinate of the point.

`y [out]`

Y-coordinate of the point.

Returns: PMIC_STATUS

Remarks This function reads 3 pairs of samples for MC13783.

24.12.2.5 PmicADCTouchStandby

This function causes the PMIC touch screen controller to enter standby mode and wait for the next pen down condition.

Prototype `PMIC_STATUS PmicADCTouchStandby(bool intEna);`

Parameters: `intEna [in]`

interrupt enable.

Returns: PMIC_STATUS

24.12.2.6 PmicADCSetComparatorThresholds

This function sets WHIGH and WLOW for automatic ADC result comparators.

Prototype `PMIC_STATUS PmicADCSetComparatorThresholds(UINT16 whigh, UINT16 wlow);`

Parameters: `whigh [in]`

a high comparator threshold.

`wlow [in]`

a low comparator threshold.

Returns: PMIC_STATUS

24.12.2.7 PmicADCGetHandsetCurrent

This function gets handset battery current measurement values.

Prototype `PMIC_STATUS PmicADCGetHandsetCurrent(RR_ADC_CONVERTOR_MODE mode, UINT16 *pResult);`

Parameters: `mode [in]`

An ADC converter mode: ADC_8CHAN_1X or ADC_1CHAN_8X

`pResult [out]`

Pointer to the handset battery current measurement value(s)

Returns: PMIC_STATUS

Remarks ADC_8CHAN_1X Mode:

This function returns one sample for battery current channel (BATTISNS).

ADC_1CHAN_8X Mode:

For MC13783, this function returns the 8 samples for battery current channel order like : ADA0=BATT; ADA1= BATT-BATTISNS; ADA2=BATT; ADA3= BATT-BATTISNS; ADA4=BATT; ADA5= BATT-BATTISNS; ADA6=BATT; ADA7= BATT-BATTISNS.

24.12.2.8 PmicADCInit

This function initializes PMIC ADC's resources.

Prototype *PMIC_STATUS PmicADCInit(void);*

Parameters: None.

Returns: PMIC_STATUS

24.12.2.9 PmicADCDeinit

This function deinitializes PMIC ADC's resources.

Prototype *void PmicADCDeinit(void);*

Parameters: None

Returns: PMIC_STATUS

24.12.3 Power Management

There is no additional power management implementation done specifically for Atlas ADC other than the implementation described in section Power Management of this document.

24.13 Backlight

24.13.1 Data types

```
typedef enum _BACKLIGHT_MODE {
    BACKLIGHT_CURRENT_CTRL_MODE,
    BACKLIGHT_TRIODE_MODE
} BACKLIGHT_MODE;

typedef enum _BACKLIGHT_CHANNEL {
    BACKLIGHT_MAIN_DISPLAY,
    BACKLIGHT_AUX_DISPLAY,
    BACKLIGHT_KEYPAD
} BACKLIGHT_CHANNEL;
```

24.13.2 Functions

24.13.2.1 PmicBacklightMasterEnable

This function sets the master enable bit of the PMIC backlight and tri-color led controller.

Prototype `PMIC_STATUS PmicBacklightMasterEnable();`

Parameters: None

Returns: PMIC_STATUS

24.13.2.2 PmicBacklightMasterDisable

This function clears the master enable bit of the PMIC backlight and tri-color led controller.

Prototype `PMIC_STATUS PmicBacklightMasterDisable();`

Parameters: None

Returns: PMIC_STATUS

24.13.2.3 PmicBacklightRampUp

This function starts backlight brightness ramp up function; ramp time is fixed at 0.5 seconds.

Prototype `PMIC_STATUS PmicBacklightRampUp(BACKLIGHT_CHANNEL channel);`

Parameters: `channel [IN]`

backlight channel

Returns: PMIC_STATUS

24.13.2.4 PmicBacklightRampDown

This function starts backlight brightness ramp down function; ramp time is fixed at 0.5 seconds.

Prototype `PMIC_STATUS PmicBacklightRampDown(BACKLIGHT_CHANNEL channel);`

Parameters: `channel[IN]`

backlight channel

Returns: PMIC_STATUS

24.13.2.5 PmicBacklightSetMode

This function sets backlight operation mode. There are two modes of operations: current control and triode mode. The Duty Cycle/Cycle Time control is retained in Triode Mode. Audio coupling is not available in Triode Mode.

Prototype `PMIC_STATUS PmicBacklightSetMode(BACKLIGHT_CHANNEL channel, BACKLIGHT_MODE mode);`

Parameters: `channel[IN]`

backlight channel

mode[IN]

backlight operation mode

Returns: PMIC_STATUS

24.13.2.6 PmicBacklightSetCurrentLevel

This function sets backlight current level. In SC55112, LED1 and LED2 are designed for a nominal full scale current of 84mA in 12mA steps. The channels are not individually adjustable, hence the channel parameter is ignored.

level	current
0	0 mA
1	12 mA
2	24 mA
3	36 mA
4	48 mA
5	60 mA
6	72 mA
7	84 mA

Prototype *PMIC_STATUS PmicBacklightSetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8 level);*

Parameters: *channel[IN]*

backlight channel

level[IN]

current level

Returns: PMIC_STATUS

24.13.2.7 PmicBacklightGetCurrentLevel

This function returns the current level for backlight channel.

Prototype *PMIC_STATUS PmicBacklightGetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8* level);*

Parameters: *channel[IN]*

backlight channel

level[OUT]

pointer to current level

Returns: PMIC_STATUS

24.13.2.8 PmicBacklightSetDutyCycle

This function sets a backlight channel duty cycle. LED perceived brightness for each zone may be individually set by setting duty cycle. The default setting is for 0% duty cycle; this keeps all zone drivers turned off even after the master enable command. Each LED current sink can be turned on and adjusted

for brightness with an independent 4 bit word for a duty cycle ranging from 0% to 100% in approximately 6.7% steps.

Prototype *PMIC_STATUS PmicBacklightSetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8 cycle);*

Parameters: *channel[IN]*

backlight channel

cycle[IN]

duty cycle

Returns: PMIC_STATUS

24.13.2.9 PmicBacklightGetDutyCycle

This function returns the duty cycle for backlight channel.

Prototype *PMIC_STATUS PmicBacklightGetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8* cycle);*

Parameters: *channel[IN]*

backlight channel

cycle[OUT]

duty cycle

Returns: PMIC_STATUS

24.13.2.10 PmicBacklightSetCycleTime

This function sets a backlight channel cycle time. Cycle Time is defined as the period of a complete cycle of Time_on + Time_off. The default Cycle Time is set to 0.01 seconds such that the 100 Hz on-off cycling is averaged out by the eye to eliminate flickering. Additionally, the Cycle Time can be programmed to intentionally extend the period of on-off cycles for a visual pulsating or blinking effect.

period	Cycle Time (sec)
0	0.01
1	0.1
2	0.5
3	2

Prototype *PMIC_STATUS PmicBacklightSetCycleTime(UINT8 period);*

Parameters: *period[IN]*

cycle time

Returns: PMIC_STATUS

24.13.2.11 PmicBacklightGetCycleTime

This function returns the cycle period for backlight controller:

Prototype *PMIC_STATUS PmicBacklightGetCycleTime(UINT8* period);*

Parameters: *period[OUT]*

pointer to the cycle time

Returns: PMIC_STATUS

24.13.2.12 PmicBacklightEnableEdgeSlow

This function enables backlight analog edge slowing mode. Analog Edge Slowing slows down the transient edges to reduce the chance of coupling LED modulation activity into other circuits. Rise and fall times will be targeted for approximately 50usec.

Prototype *PMIC_STATUS PmicBacklightEnableEdgeSlow();*

Parameters: None

Returns: PMIC_STATUS

24.13.2.13 PmicBacklightDisableEdgeSlow

This function disables backlight analog edge slowing mode. The backlight drivers will default to an "Instant On" mode.

Prototype *PMIC_STATUS PmicBacklightDisableEdgeSlow();*

Parameters: None

Returns: PMIC_STATUS

24.13.3 Power Management

There is no additional power management implementation done specifically for Atlas Backlight and Atlas Tri-Color other than the implementation described in section Power Management of this document.

24.14 Connectivity

The Connectivity module in the PMIC provides transceiver circuitry to connect USB, RS-232 or CEA-936 type of devices to the USB OTG mini-AB receptacle on the PMIC board. The Connectivity API provides ways to configure the transceiver. This chapter provides information about the service API provided by Connectivity module.

24.14.1 Data types

```
// General data types
// Callback function prototype
typedef void (*PMIC_CONVITY_CALLBACK)(const PMIC_CONVITY_EVENTS);

// handle for PMIC access
typedef long PMIC_CONVITY_HANDLE;

// Operating modes of connectivity interface
typedef enum {
      USB = 0,
```

```
      RS232 = 1,
      CEA936_MONO = 4,
      CEA936_STEREO = 5,
      CEA936_TEST_RIGHT = 6,
      CEA936_TEST_LEFT = 7
} PMIC_CONVITY_MODE;

// Connectivity interface events. Bitwise or-able
typedef enum {
      USB_DETECT_4V4_RISE         = 1,     // VBUS arose to cross 4.4v
      USB_DETECT_4V4_FALL         = 2,     // VBUS fell below 4.4v
      USB_DETECT_2V0_RISE         = 4,     // VBUS arose to cross 2.0v
      USB_DETECT_2V0_FALL         = 8,     // VBUS fell below 2.0v
      USB_DETECT_0V8_RISE         = 16,    // VBUS arose to cross 0.8v
      USB_DETECT_0V8_FALL         = 32,    // VBUS fell below 0.8v
      USB_DETECT_MINI_A           = 64,    // mini-A plug connected
      USB_DETECT_MINI_B           = 128,   // mini-B plug connected
      USB_DETECT_NON_USB_ACCESORY = 256,   // non-USB device connected
      USB_DETECT_FACTORY_MODE     = 512,   // factory mode
      USB_DETECT_SE1_RISE         = 1024,  // single ended 1 appeared/detected
      USB_DETECT_SE1_FALL         = 2048,  // single ended 1 disappeared/no more present
      USB_DETECT_CKDETECT         = 4096   // car-kit connected
} PMIC_CONVITY_EVENTS;

// USB mode specific data types
// USB transceiver speed
typedef enum {
      USB_LOW_SPEED,
      USB_FULL_SPEED,
      USB_HIGH_SPEED
} PMIC_CONVITY_USB_SPEED;

// Modes as USB transceiver
typedef enum {
      USB_HOST,
      USB_PERIPHERAL
} PMIC_CONVITY_USB_MODE;

// input source for USB transceiver's power regulator
typedef enum {
      USB_POWER_VBUS = 1,
      USB_POWER_INTERNAL = 2,
      USB_POWER_INTERNAL_BOOST = 0
} PMIC_CONVITY_USB_POWER_IN;

// output voltage of USB transceiver's power regulator
typedef enum {
      USB_POWER_2V775 = 0,
      USB_POWER_3V3
} PMIC_CONVITY_USB_POWER_OUT;

// USB transceiver operating mode
typedef enum {
      USB_TRANSCEIVER_OFF,
      USB_SINGLE_ENDED_UNIDIR_TX,
      USB_SINGLE_ENDED_UNIDIR_RX,
      USB_SINGLE_ENDED_BIDIR_TX,
```

```
      USB_SINGLE_ENDED_BIDIR_RX,
      USB_SINGLE_ENDED_LOW,
      USB_DIFFERENTIAL_UNIDIR_TX,
      USB_DIFFERENTIAL_UNIDIR_RX,
      USB_DIFFERENTIAL_BIDIR_TX,
      USB_DIFFERENTIAL_BIDIR_RX,
      USB_SUSPEND_ON,
      USB_SUSPEND_OFF,
      USB_OTG_SRP_DLP_START,
      USB_OTG_SRP_DLP_STOP
} PMIC_CONVITY_USB_TRANSCEIVER_MODE;

// USB on-the-go configuration. Bitwise or-able
typedef enum {
      USB_OTG_SE0CONN                = 1,
      USB_OTG_DLP_SRP                = 2,
      USB_PULL_OVERRIDE              = 4,
      USB_VBUS_CURRENT_LIMIT_HIGH    = 8,
      USB_VBUS_CURRENT_LIMIT_LOW     = 16,
      USB_VBUS_CURRENT_LIMIT_LOW_10MS = 32,
      USB_VBUS_CURRENT_LIMIT_LOW_20MS = 64,
      USB_VBUS_CURRENT_LIMIT_LOW_30MS = 128,
      USB_VBUS_CURRENT_LIMIT_LOW_40MS = 256,
      USB_VBUS_CURRENT_LIMIT_LOW_50MS = 512,
      USB_VBUS_CURRENT_LIMIT_LOW_60MS = 1024,
      USB_VBUS_PULLDOWN              = 2048
} PMIC_CONVITY_USB_OTG_CONFIG;

// USB device type, matching the connector connected to the receptacle.
typedef enum {
      USB_A_DEVICE,
      USB_B_DEVICE
} PMIC_CONVITY_USB_DEVICE_TYPE;

// RS-232 mode specific data types
// RS-232 transceiver external connections
typedef enum {
      RS232_TX_UDM_RX_UDP,
      RS232_TX_UDP_RX_UDM,
      RS232_TX_RX_EXTERNAL_DEFAULT
} PMIC_CONVITY_RS232_EXTERNAL;

// RS-232 transceiver internal connections
typedef enum {
      RS232_TX_USE0VM_RX_UDATVP,
      RS232_TX_UDATVP_RX_URXVM,
      RS232_TX_UTXDI_RX_URXDO,
      RS232_TX_RX_INTERNAL_DEFAULT
} PMIC_CONVITY_RS232_INTERNAL;

// CEA-936 mode specific data types
// Accessory detection config. Bitwise or-able
typedef enum {
      ACCESSORY_ID_ID100KPU = 1,
      ACCESSORY_ID_IDPUCNTRL = 2,
      ACCESSORY_ID_DP150KPU = 4
} PMIC_CONVITY_CEA936_DETECTION_CONFIG;
```

```
// CEA-936 mode exit signals
typedef enum {
      CEA936_UID_NO_PULLDOWN,
      CEA936_UID_PULLDOWN_6MS,
      CEA936_UID_PULLDOWN,
      CEA936_UDMPULSE
} PMIC_CONVITY_CEA936_EXIT_SIGNAL;
```

24.14.2 Functions

24.14.2.1 PmicConvityOpen

Attempts to open and gain exclusive access to the PMIC connectivity hardware. An initial operating mode (e.g., USB or RS-232) must be specified.

If the open request is successful, then a numeric handle is returned and this handle must be used in all subsequent calls. The same handle must also be used in the close call when use of the PMIC connectivity hardware is no more required.

The open request will fail if another thread has already obtained the device handle and has not yet called PmicConvityClose().

Prototype *PMIC_STATUS PmicConvityOpen (PMIC_CONVITY_HANDLE *const handle, const PMIC_CONVITY_MODE mode);*

Parameters *handle [out]*

device handle to be used in subsequent PMIC connectivity API calls.

mode [in]

initial mode to be set.

Returns PMIC_STATUS

24.14.2.2 PmicConvityClose

Terminates further access to PMIC connectivity hardware. This also allows another thread to successfully call PmicConvityOpen().

Prototype *PMIC_STATUS PmicConvityClose (const PMIC_CONVITY_HANDLE handle);*

Parameters *handle [in]*

device handle from PmicConvityOpen() call.

Returns PMIC_STATUS

24.14.2.3 PmicConvitySetMode

Change the current operating mode of the PMIC Connectivity hardware. The operating mode will be set to the requested mode if it supported by the hardware; otherwise PMIC_NOT_SUPPORTED is returned.

Prototype *PMIC_STATUS PmicConvitySetMode (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_MODE mode);*

Parameters *handle [in]*

Device handle from PmicConvityOpen() call.

mode [in]

New mode to be set.

Returns PMIC_STATUS

24.14.2.4 PmicConvityGetMode

This function gets the PMIC connectivity hardware's current operating mode.

Prototype
```
PMIC_STATUS PmicConvityGetMode (const PMIC_CONVITY_HANDLE handle,
PMIC_CONVITY_MODE *const mode);
```

Parameters *handle [in]*

device handle from PmicConvityOpen() call.

mode [out]

current mode.

Returns PMIC_STATUS

24.14.2.5 PmicConvityReset

This function resets all the hardware registers to the initial power-on/reset state.

Prototype
```
PMIC_STATUS PmicConvityReset (const PMIC_CONVITY_HANDLE handle);
```

Parameters *handle [in]*

device handle from PmicConvityOpen() call.

Returns PMIC_STATUS

24.14.2.6 PmicConvitySetCallback

This function is used to register callback for receiving notifications related to connectivity interface related events. For example, the USB subsystem could use this to detect connect/disconnect of USB connector.

Prototype
```
PMIC_STATUS PmicConvitySetCallback (const PMIC_CONVITY_HANDLE handle,
const PMIC_CONVITY_CALLBACK func, const PMIC_CONVITY_EVENTS eventMask);
```

Parameters *handle [in]*

device handle from PmicConvityOpen() call.

func [in]

function pointer of callback function.

eventMask [in]

events to be notified through callback.

Returns PMIC_STATUS

24.14.2.7 PmicConvityClearCallback

This function deregisters the callback that was previously registered using *PmicConvitySetCallback()*.

Prototype `PMIC_STATUS PmicConvityClearCallback (const PMIC_CONVITY_HANDLE handle);`

Parameters `handle [in]` device handle from PmicConvityOpen() call.

Returns PMIC_STATUS

24.14.2.8 PmicConvityGetCallback

This function returns the callback function and event mask which is currently set.

Prototype
```
PMIC_STATUS PmicConvityGetCallback (const PMIC_CONVITY_HANDLE handle,
PMIC_CONVITY_CALLBACK *const func, PMIC_CONVITY_EVENTS *const eventMask);
```

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`func [out]` function pointer of registered callback function.

`eventMask [out]` registered events bitmask.

Returns PMIC_STATUS

24.14.2.9 PmicConvityGetEventStatus

This function can be used to get the events which are signalled (asserted) currently in the PMIC connectivity.

Prototype
```
PMIC_STATUS PmicConvityGetEventStatus (const PMIC_CONVITY_HANDLE handle,
PMIC_CONVITY_EVENTS *const events);
```

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`events [out]` bit mask of currently signalled events.

Returns PMIC_STATUS.

24.14.2.10 PmicConvityUsbSetSpeed

This function sets the USB transceiver speed and USB OTG device mode. Upon success, the USB transceiver is enabled. In case of USB_HOST, the VBUS is also enabled.

High speed is not supported by MC13783.

Prototype
```
PMIC_STATUS PmicConvityUsbSetSpeed (const PMIC_CONVITY_HANDLE handle,
const PMIC_CONVITY_USB_SPEED speed, const PMIC_CONVITY_USB_MODE  mode);
```

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`speed [in]` transceiver speed.

`mode [in]` host / peripheral mode of the OTG interface.

Returns PMIC_STATUS.

24.14.2.11 PmicConvityUsbGetSpeed

This function gets the USB transceiver speed.

Prototype *PMIC_STATUS PmicConvityUsbGetSpeed (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_USB_SPEED *const speed, PMIC_CONVITY_USB_MODE *const mode);*

Parameters
handle [in] device handle from PmicConvityOpen() call.
speed [out] transceiver speed.
mode [out] host / peripheral mode of the OTG interface.

Returns PMIC_STATUS

24.14.2.12 PmicConvityUsbSetPowerSource

This function sets input source and output voltage level for the USB transceiver's power supply. Besides setting the input source and output voltage level, this API enables the regulator.

Prototype *PMIC_STATUS PmicConvityUsbSetPowerSource (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_POWER_IN pwrin, const PMIC_CONVITY_USB_POWER_OUT pwrout);*

Parameters
handle [in] device handle from PmicConvityOpen() call.
pwrin [in] input source for USB power regulator.
pwrout [in] output voltage level of the USB power regulator.

Returns PMIC_STATUS

24.14.2.13 PmicConvityUsbGetPowerSource

This function gets settings for the USB transceiver's power supply.

Prototype *PMIC_STATUS PmicConvityUsbGetPowerSource (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_USB_POWER_IN *const pwrin, PMIC_CONVITY_USB_POWER_OUT *const pwrout);*

Parameters
handle [in] device handle from PmicConvityOpen() call.
pwrin [out] input source for USB power regulator.
pwrout [out] output voltage level of the USB power regulator.

Returns PMIC_STATUS.

24.14.2.14 PmicConvityUsbSetXcvr

This function sets the USB transceiver's operating mode. Implementation for MC13783 has the following behaviour:

1. Setting any one of the following pairs have the same effect, as TX and RX cannot be controlled at MC13783:
   - USB_SINGLE_ENDED_UNIDIR_TX and USB_SINGLE_ENDED_UNIDIR_RX
   - USB_SINGLE_ENDED_BIDIR_TX and USB_SINGLE_ENDED_BIDIR_RX
   - USB_DIFFERENTIAL_UNIDIR_TX and USB_DIFFERENTIAL_UNIDIR_RX

- USB_DIFFERENTIAL_BIDIR_TX and USB_DIFFERENTIAL_BIDIR_RX

2. The USB_SINGLE_ENDED_LOW mode has no effect, as making the outputs as SE0 cannot be done by MC13783.

Prototype *PMIC_STATUS PmicConvityUsbSetXcvr (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_TRANSCEIVER_MODE mode);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

mode [in] desired mode of USB transceiver. Only one of the modes defined in PMIC_CONVITY_USB_TRANSCEIVER_MODE can be set at a time.

Returns PMIC_STATUS.

24.14.2.15 PmicConvityUsbGetXcvr

This function gets the USB transceiver's operating mode.

Prototype *PMIC_STATUS PmicConvityUsbGetXcvr (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_USB_TRANSCEIVER_MODE *const mode);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

mode [out] mode of USB transceiver.

Returns PMIC_STATUS

24.14.2.16 PmicConvityUsbOtgSetDlpDuration

This function sets the data line pulsing duration for USB OTG session request protocol. This function will return PMIC_NOT_SUPPORTED for MC13783, as DLP duration for MC13783 has been fixed at 7.5ms, and is not programmable.

Prototype *PMIC_STATUS PmicConvityUsbOtgSetDlpDuration (const PMIC_CONVITY_HANDLE handle, const unsigned int duration);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

duration[in] duration of DLP (milliseconds).

Returns PMIC_STATUS.

24.14.2.17 PmicConvityUsbOtgGetDlpDuration

This function gets the data line pulsing duration for USB OTG session request protocol. This function will return PMIC_NOT_SUPPORTED for MC13783, as DLP duration for MC13783 has been fixed at 7.5ms, and is not programmable.

Prototype *PMIC_STATUS PmicConvityUsbOtgGetDlpDuration (const PMIC_CONVITY_HANDLE handle, unsigned int *const duration);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

duration [out] duration of DLP.

Returns PMIC_STATUS

24.14.2.18 PmicConvityUsbOtgBeginHnp

This function sets up the signaling required to begin the host negotiation protocol (HNP). This function explicitly begins HNP by setting up D+ pull-up on A device and disconnecting all pull-up and pull-down on B device.

Prototype `PMIC_STATUS PmicConvityUsbOtgBeginHnp(const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_DEVICE_TYPE type);`

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`type [in]` USB on-the-go device type determined by the type of connector presently connected to the receptacle.

Returns PMIC_STATUS.

24.14.2.19 PmicConvityUsbOtgEndHnp

This function sets up the signaling required to end the host negotiation protocol (HNP). This function explicitly ends the HNP by disconnecting pull-up on A device and enabling pull-up on B device.

Prototype `PMIC_STATUS PmicConvityUsbOtgEndHnp(const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_DEVICE_TYPE type);`

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`type [in]` USB on-the-go device type determined by the type of connector presently connected to the receptacle.

Returns PMIC_STATUS.

24.14.2.20 PmicConvityUsbOtgSetConfig

This function sets the USB On-The-Go configuration. Multiple configuration settings may be OR-ed together. Selecting conflicting settings (e.g., multiple VBUS current limits) will result in undefined behavior.

Prototype `PMIC_STATUS PmicConvityUsbOtgSetConfig (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_OTG_CONFIG cfg);`

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`cfg [in]` on-the-go configuration.

Returns PMIC_STATUS.

24.14.2.21 PmicConvityUsbOtgClearConfig

This function clears the USB On-The-Go configuration. Multiple configuration settings may be OR-ed together.

Prototype `PMIC_STATUS PmicConvityUsbOtgClearConfig (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_USB_OTG_CONFIG cfg);`

Parameters `handle [in]` device handle from PmicConvityOpen() call.

`cfg [in]` on-the-go configuration.

Returns PMIC_STATUS.

24.14.2.22 PmicConvityUsbOtgGetConfig

This function gets the USB On-The-Go configuration.

Prototype *PMIC_STATUS PmicConvityUsbOtgGetConfig (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_USB_OTG_CONFIG *const cfg);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

cfg [out] on-the-go configuration.

Returns PMIC_STATUS.

24.14.2.23 PmicConvityRs232SetConfig

This function sets the connectivity interface to the selected RS-232 operating mode.

Prototype *PMIC_STATUS PmicConvityRs232SetConfig (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_RS232_INTERNAL cfgInternal, const PMIC_CONVITY_RS232_EXTERNAL cfgExternal, const BOOL txTristated);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

cfgInternal [in] transceiver internal connection configuration

cfgExternal [in] transceiver external connection configuration

txTristate [in] true: tristate the Tx line on cable side; false: activate the Tx line on cable side.

Returns PMIC_STATUS.

24.14.2.24 PmicConvityRs232GetConfig

Get the connectivity interface's current RS-232 operating mode.

Prototype *PMIC_STATUS PmicConvityRs232GetConfig (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_RS232_INTERNAL *const cfgInternal, PMIC_CONVITY_RS232_EXTERNAL *const cfgExternal, BOOL *const txTristated);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

cfgInternal [out] transceiver internal connection configuration

cfgExternal [out] transceiver external connection configuration

txTristate[out] true: Tx line on cable side is tri-stated; false: Tx line on cable side is active.

Returns PMIC_STATUS.

24.14.2.25 PmicConvityCea936SetDetectionConfig

This function sets the circuitry for accessory type identification. In MC13783, this API can be used to set-up the ID100KPU, IDPUCNTRL and DP150KPU pull-up resistor configuration which helps in accessory type detection.

Prototype *PMIC_STATUS PmicConvityCea936SetDetectionConfig (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_CEA936_DETECTION_CONFIG cfg);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

cfg [in] circuitry set-up for accessory type identification

Returns PMIC_STATUS.

24.14.2.26 PmicConvityCea936GetDetectionConfig

This function gets the current circuitry set-up for accessory type identification.

Prototype *PMIC_STATUS PmicConvityCea936GetDetectionConfig (const PMIC_CONVITY_HANDLE handle, PMIC_CONVITY_CEA936_DETECTION_CONFIG *const cfg);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

cfg [out] circuitry set-up for accessory type identification

Returns PMIC_STATUS.

24.14.2.27 PmicConvityCea936ExitSignal

This function is used to signal the accessory to exit the audio mode.

Prototype *PMIC_STATUS PmicConvityCea936ExitSignal (const PMIC_CONVITY_HANDLE handle, const PMIC_CONVITY_CEA936_EXIT_SIGNAL signal);*

Parameters *handle [in]* device handle from PmicConvityOpen() call.

signal [in] exit signaling method

Returns PMIC_STATUS

24.14.3 Power Management

There is no additional power management implementation done specifically for Atlas Connectivity other than the implementation described in section Power Management of this document.

Chapter 25
Power Manager

The Power Manager module is used to help control the power efficiency of the system. Power Manager provides a framework that provides interface to application programs, control of peripheral device power states and allows peripheral devices to self manage their power state.

25.1 Power Manager Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	N/A
Import Library	N/A
Driver DLL	Pm.dll
Catalog Item	Core OS −>Windows CE Devices −>Core OS Services −>Power Management −> Power Management (Full)
SYSGEN Dependency	SYSGEN_PM
BSP Environment Variables	N/A

The default power manager used in the BSP is located at WINCE500\PUBLIC\COMMON\OAK\DRIVERS\PM.

25.2 Requirements

Include and test the power manager provided in the Platform Builder public directory.

25.3 Hardware Operation

Power Manager does not interface directly to peripheral devices.

25.4 Software Operation

The Platform Builder help documents the power manager framework and sample power manager. See **Developing a Device Driver −> Power Management**.

The system power states implemented in the sample power manager are documented in Platform Builder help at **Developing a Device Driver –> Power States –> System Power States –>Example System Power States.**

25.4.1 Power Management

Not applicable.

25.4.2 Registry Settings

In this system power state, the user is interacting actively with the system.

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\On]
    "Default"=dword:0           ; D0
    "Flags"=dword:10000         ; POWER_STATE_ON

 In this system power state, the user may be interacting with the
 system, but not actively.  For instance, they might be looking at
 the screen or they might not.  In this power state the system is
 "idle" but still in use by the user, so all devices still be
 operational (but possibly with some latency).

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\UserIdle]
    "Default"=dword:1           ; D1
    "Flags"=dword:0

 In this system power state, the user is not considered to be using
 the system, even passively.  However, the system is not suspended
 and system programs may be doing work on the user's behalf.  In this
 power state the system is "idle" but might still be used by system
 programs.  Devices that aren't actively doing work might be powered
 down.

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\SystemIdle]
    "Default"=dword:2           ; D2
    "Flags"=dword:0

 In this system power state, the system is suspended.  Devices are turned
 off, interrupts are not being serviced, and the CPU is stopped.
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Suspend]
    "Default"=dword:3           ; D3
    "Flags"=dword:200000        ; POWER_STATE_SUSPEND

 Entering this system power state reboots the system with a clean object
 store.  If an OEM includes this state in their platform, they must
 support KernelIoControl() with IOCTL_HAL_REBOOT.

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\ColdReboot]
    "Default"=dword:4           ; D4
    "Flags"=dword:800000        ; POWER_STATE_RESET

 Entering this system power state reboots the system.  If an OEM includes this state in
 their platform, they must support KernelIoControl() with IOCTL_HAL_REBOOT.
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\Reboot]
    "Default"=dword:4           ; D4
    "Flags"=dword:800000        ; POWER_STATE_RESET

 Entering this system power state shuts down the system.  All devices are powered off,
 resuming may require user intervention.  The system will cold boot on resume.
 Supporting this power state requires that the OEM customize the Power Manager
 to recognize POWER_STATE_OFF and take platform-specific action to remove
 power.
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\State\ShutDown]
    "Default"=dword:4           ; D4
    "Flags"=dword:20000         ; POWER_STATE_OFF

Default Activity Timers

 These registry values set up activity timers inside the Power Manager.
 GWES and/or other system components need to reset them periodically
 to keep the associated inactivity event from being set.

 Defining timers causes the PM to create a set of named events for resetting
 the timer and for obtaining its activity status.  See the PM documentation
 for more information.

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\ActivityTimers\UserActivity]
    "Timeout"=dword:1                       ; in seconds

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\ActivityTimers\SystemActivity]
    "Timeout"=dword:1                       ; in seconds


 Default System Power State Transition Timeouts

 These registry values configure the interval of time the Power Manager allows
 to pass during periods of inactivity before updating the system power state.

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Power\Timeouts]
    "ACUserIdle"=dword:3c           ; 60 seconds
    "ACSystemIdle"=dword:12c        ; 300 seconds
    "ACSuspend"=dword:0             ; 0 seconds

    "BattUserIdle"=dword:3c         ; 60 seconds
    "BattSystemIdle"=dword:b4       ; 180 seconds
    "BattSuspend"=dword:12c         ; 300 seconds
```

25.5 Unit Test

The power applet in the control panel is used to test power manager. The timer settings to transition between system power states can be adjusted and proper behavior can be observed. The change in the capacity of the battery is shown in the power applet by querying the Battery driver. Details regarding the battery driver can be looked in chapter 5.

25.6 Power Manager API Reference

The Power Manager interfaces with applications and device drivers. The following sections provide reference to the definition of these program interfaces.

25.6.1 Application Interface

The power manager API's for applications are documented in help at:

- **Developing a Device Driver –> Power Management –> Power Manager Interfaces –> Application Interface**
- **Developing a Device Driver –> Power Management –> Power Manager Interfaces –> Notification Interface**
- **Developing a Device Driver –> Power Management –> Power Management Reference**

25.6.2 Device Driver Interface

The interface for device drivers is documented in help at:

- **Developing a Device Driver –> Power Management –> Power Manager Interfaces –> Device Driver Interface**
- **Developing a Device Driver –> Power Management –> Power Management Reference**

Chapter 26
Pulse-Width Modulator (PWM)

The Pulse Width Modulator (PWM) module is used to generate 16-bit resolution sound from sample audio images and can also be used to generate tones.

26.1 PWM Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\PWM
CSP Driver Path	..\CSP\ARM\FREESCALE\<*TGTSOC*>\DRIVERS\PWM
CSP Static Library	<*TGTSOC*>_pwm.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\PWM
Import Library	N/A
Driver DLL	pwm.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> PWM
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PWM =1

26.2 Requirements

The PWM driver should meet the following requirements:

1. The driver shall be a stream interface driver implementing the programming interface defined in this document.
2. The driver shall support two power management modes, full on and full off.

26.3 Hardware Operation

Refer to the chapter on PWM in the hardware specification document for detailed operation and programming information.

26.3.1 Conflicts with other SoC peripherals

26.3.1.1 MX31/MX32 Peripheral Conflicts

Since the PWMO pin that connects to the Buzzer is multiplexed with the ATA_IORDY pin of the ATA, it will not be possible to include and use both ATA and PWM modules in the same OS image.

In order to connect the PWM output to the buzzer, it is required to turn ON the buzzer enable switch in the Base board SW2 switch settings.

26.4 Software Operation

26.4.1 Communicating with the PWM

The PWM is a stream interface driver, and is thus accessed through the file system APIs. To generate signals using the PWM, 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 basic steps are detailed below.

26.4.2 Creating a Handle to the PWM

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

To open a handle to the PWM

1. Insert a colon after the PWM port for the first parameter, *lpFileName*.
   — For example, specify PWM1: as the PWM port.
2. Specify 0 in the *dwShareMode* parameter. PWM port cannot be shared.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter.
   — This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

The following code example shows how to open a PWM port.

```
g_hDriver =  CreateFile(_T("PWM1:"),      // name of device
             GENERIC_READ|GENERIC_WRITE,  // access (read-write) mode
             0,                           // sharing mode
             NULL,                        // security attributes (ignored)
             OPEN_EXISTING,               // creation disposition
             FILE_FLAG_RANDOM_ACCESS,     // flags/attributes
             NULL);                       // template file (ignored)
```

26.4.3 Configuring the PWM

N/A

26.4.4 Write Operations

Once the handle to the driver is obtained, it is possible to generate the PWM sound output by writing input PWM-sample data in a specific format. The PWM-sample information written to PWM driver contains the 32bit sample data, 32bit period value and the duration in milliseconds. The format of the PWM-sample structure is given below along with the code example. With the current driver implementation only one sample can be fed at a time through the WriteFile Operation.

```
/* struct of pwm play sample */
struct {
   UINT32  sample;              // pwm sample value
   UINT32  period;              // pwm period
   UINT32  duration;            // duration of the sample (in millisecs)
} PwmSample = {1, 1, 8000};
```

```
if(!WriteFile(g_hDriver, &PwmSample, sizeof(PwmSample), (LPDWORD) &bytesWritten,
NULL))
{
   g_pKato->Log(PWM_ZONE_ERROR,TEXT("PWMTest.cpp:PWMRegisterTest():WriteFile
                    failed!\r\n"));
        Debug(TEXT("PWMTest.cpp:PWMRegisterTest(): WriteFile failed!\r\n"));
        return TPR_FAIL;
}
```

Here, the **PwmSample** parameter contains the PWM-Sample data. The above code will generate a 8second human audible tone in the buzzer.

26.4.5 Read Operations

The read operation is performed using ReadFile function. Read operation in PWM driver is used to read the contents of the following registers of PWM in order.

PWM Control Register,

PWM Status Register,

PWM Interrupt Register,

PWM Sample Register,

PWM Period Register

PWM Counter Register.

The following code example shows a typical read operation.

```
      UINT32 PwmReadRegs[6]; //Buffer to store the register contents

if(!ReadFile(g_hDriver, PwmReadRegs, sizeof(PwmReadRegs), (LPDWORD) &bytesRead,
NULL))
{
```

```
            g_pKato->Log(PWM_ZONE_ERROR, TEXT("PWMTest.cpp:PWMRegisterTest(): ReadFile
                         failed!\r\n"));
             Debug(TEXT("PWMTest.cpp:PWMRegisterTest(): ReadFile failed!\r\n"));
             return TPR_FAIL;
}
```

26.4.6 Closing the Handle to the PWM

Call the CloseHandle function to close a handle to the PWM when an application is done using it.

CloseHandle has one parameter, which is the handle returned by the CreateFile function call that opened the PWM port.

26.4.7 Power Management

The primary method for limiting power consumption in the PWM module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. PWM module clock is enabled during the initialization stage. This is done to reset the PWM module during initialization. This clock is disabled whenever there is a Power Down sequence happening through the IOCTL_POWER_SET Ioctl call. Once disabled, the clock is again enabled by using the same IOCTL_POWER_SET Ioctl call during Power Up sequence.

26.4.7.1 PowerUp

This function will do a reset of the PWM Module.

26.4.7.2 PowerDown

This function is not implemented for the PWM driver.

26.4.7.3 IOCTL_POWER_CAPABILITIES

We advertise the power management capabilities with power manager through this IOCTL. The PWM module supports only two power states: D0 and D4.

26.4.7.4 IOCTL_POWER_SET

This IOCTL requests a change from one device power state to another. D0 and D4 are the only two supported **CEDEVICE_POWER_STATE** in PWM driver. So any power change request to the driver to a value of D3 or D4 will result in the clock being disabled for the PWM Module, while for a value of D0 or D1 will result in the clock getting enabled.

26.4.7.5 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 will not generally issue an **IOCTL_POWER_GET** call to the device unless an application calls **GetDevicePower** with the POWER_FORCE flag set.

26.4.8 PWM Registry Settings

The following registry key is required to be added in the platform.reg file to properly load the PWM driver.

```
IF BSP_PWM
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWM]
   "Prefix"="PWM"
   "Dll"="pwm.dll"
   "Index"=dword:1
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
ENDIF
```

26.5 Unit Test

The PWM tests verify that the PWM driver properly initializes and controls the PWM controller.

26.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Digital Oscilloscope	PWMWaveform Test requires the usage of digital oscilloscope to measure the frequency of the generated waveform and compare it with the theoretically calculated value.

26.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
pwmtest.dll	Test .dll file

26.5.3 Building the PWM Tests

In order to build the PWM tests, complete the following steps:

1. Build an OS image for the desired configuration.
2. Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
3. Change to the PWM Tests directory. (\WINCE500\SUPPORT\TESTS\PWM)
4. Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
5. Enter the build command at the prompt and press return.

After the build completes, the pwmtest.dll file will be located in the $(_FLATRELEASEDIR) directory.

26.5.4 Running the PWM Tests

The command line for running the PWM tests is tux –o –d pwmtest. The PWM tests do not contain any test specific command line options.

The following table describes the test cases contained in the PWM tests.

Test Case	Description	Parameters
100: PWM Reset Test	Verifies that the PWM software reset happens properly.	None
1000: PWM Register Test	This test verifies the possibilities of register read and write functions successfully on PWM registers.	None
1001: PWM Waveform Test	This test verifies wether the frequency of the generated PWM waveforms matches with that of the theoretical values. The frequency of the generated waveforms needs be verified manually using Oscilloscope.	None

26.6 PWM Driver API Reference

26.6.1 PWM Driver IOCTLS

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

26.6.1.1 PWM_IOCTL_RESET

This **DeviceIoControl** request disables the PWM, performs a software reset, disables all interrupts and clears interrupt status bits.

Parameters None

26.6.1.2 IOCTL_POWER_CAPABILITIES

This IOCTL request is used to determine the power capabilities of the device.

Parameters

lpOutBuffer Pointer to memory location that holds the POWER_CAPABILITIES structure that determines the power capabilities of the device. Refer Platform Builder documentation for the description of this structure.

26.6.1.3 IOCTL_POWER_GET

This **DeviceIoControl** request retrieves the current power state information.

Parameters

lpOutBuffer Pointer to memory location where the current power state expressed as CEDEVICE_POWER_STATE is stored. Refer Platform Builder documentation for the description of this CEDEVICE_POWER_STATE enumeration.

26.6.1.4 IOCTL_POWER_SET

This **DeviceIoControl** request sets a new power state to the device.

Parameters

lpOutBuffer Pointer to the memory location where the new power state expressed as CEDEVICE_POWER_STATE is stored. Refer Platform Builder documentation for the description of this CEDEVICE_POWER_STATE enumeration.

26.6.2 PWM Driver Macros

N/A

26.6.3 PWM Driver Structures

N/A

Chapter 27
Secure Digital Host Controller

The Secure Digital Host Controller (SDHC) module supports Multimedia Cards (MMC), Secure Digital Cards (SD) and Secure Digital I/O and Combo Cards (SDIO). MX31 and MX32 have two SDHC hardware modules. One host controller supports only one card to be connected to it.

The SDHC driver provides the interface between Microsoft's SD Bus driver and the SDHC hardware.

27.1 SDHC Driver Summary

27.1.1 MX31 & MX32 SDHC Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31,MX32ADS
Target SOC (TGTSOC)	MX31,MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\SDHC
CSP Driver Path	N/A
CSP Static Library	mxarm11_sdhc.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\SDHC
Import Library	N/A
Driver DLL	sdhc.dll
Catalog Item	Third Party −>BSPs −>Freescale <*TGTPLAT*> −>Device Drivers −> SD Controller −>SD Host Controller 1 Third Party −>BSPs −>Freescale <*TGTPLAT*> −>Device Drivers −> SD Controller −>SD Host Controller 2
SYSGEN Dependency	SYSGEN_SD_MEMORY=1 SYSGEN_BTH=1 SYSGEN_BTH_SDIO_ONLY=1 BSP_NIC_WLAN6060_SDIO=1 IMGSDBUS2=1
BSP Environment Variables	BSP_SDHC1=1 BSP_SDHC2=1

27.2 Requirements

The SDHC driver should meet the following requirements:

1. The driver shall support the MX31 ADS Secure Digital Host Controllers.
2. The driver shall support the MX32 ADS Secure Digital Host Controllers.
3. The driver shall support two Host Controllers to be functional at the same time.
4. The driver shall support Low capacity MMC, Low and High capacity SD and SDIO cards.
5. The driver shall support Power Management modes, full on and full off only.

27.3 Hardware Operation

Refer to the chapter on the Secure Digital Host Controller (SDHC) in the hardware specification document for detailed operation and programming information.

27.3.1 Conflicts with other SoC peripherals

27.3.1.1 MX31 & MX32 Peripheral Conflicts

In Alternate Mode 1, SDHC1 conflicts with Memory Stick (MS1). Configure the pins in Functional Mode to activate SDHC1 signals.

In Functional Mode, SDHC2 conflicts with PCMCIA. Configure the pins in Alternate Mode 1 to activate SDHC2 signals.

27.4 Software Operation

The SDHC driver follows the Microsoft-recommended architecture for Secure Digital Host Controller drivers. 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 Microsoft documentation at the following URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddk5/html/wce50conSecureDigitalCardDriverDevelopmentConcepts.asp

27.4.1 Required Catalog Items

27.4.1.1 SD and MMC memory card support

Catalog –> Device Drivers –> SDIO –> SD Memory.

27.4.1.2 Bluetooth SDIO Support

Catalog –> Core OS –> Communication Services and Networking –> Networking –> Personal Area Network –>Bluetooth –> Bluetooth Protocol Stack with Transport Driver Support –> Bluetooth Stack with Universal Loadable Driver

Catalog –> Core OS –> Communication Services and Networking –> Networking –> Personal Area Network > Bluetooth –> Bluetooth Protocol Stack with Transport Driver Support –> Bluetooth Stack with Integrated Driver

27.4.1.3 Wi-Fi SDIO Support

Catalog –> Device Drivers –> SDIO –> SD Memory –> SDIO WiFi

27.4.2 SDHC Registry Settings

27.4.2.1 MX31 & MX32 SDHC Register Settings

The following registry keys are required to properly load the SDHC driver.

```
#if (defined BSP_SDHC1 || defined BSP_SDHC2)
[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Class\SDMemory_Class]
   "BlockTransferSize"=dword:100  ; Overwrite from default 64 blocks.
;   "SingleBlockWrites"=dword:1  ; alternatively force the driver to use single block
access

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Class\MMC_Class]
  "BlockTransferSize"=dword:100  ; Overwrite from default 64 blocks.
;  "SingleBlockWrites"=dword:1  ; alternatively force the driver to use single block
access

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

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

IF BSP_SDHC1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SDHC_ARM11_1]
    "Order"=dword:21
    "Dll"="sdhc.dll"
    "Prefix"="SDH"
    "ControllerISTPriority"=dword:64
    "Index"=dword:1
ENDIF ;BSP_SDHC1

IF BSP_SDHC2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SDHC_ARM11_2]
    "Order"=dword:21
    "Dll"="sdhc.dll"
    "Prefix"="SDH"
    "ControllerISTPriority"=dword:64
    "Index"=dword:2
ENDIF ;BSP_SDHC2
```

27.4.3 DMA Support

27.4.3.1 MX31, MX32 DMA Support

SDHC driver supports DMA mode and non-DMA mode of data transfer. The driver defaults to DMA mode of transfer always. The driver does not allocate or manage DMA buffers internally. All buffers are allocated and managed by the upper layers, the details of which are given in the request submitted to the driver. For every request submitted to it, the driver attempts to build a DMA Scatter Gather Buffer Descriptor list for the buffer passed to it by the upper layer. For cases where this list cannot be built, the driver falls back to the non-DMA mode of transfer. The default configuration is maintained in the file bsp_cfg.h using the parameters BSP_SDMA_SUPPORT_SDHC1 and BSP_SDMA_SUPPORT_SDHC2. A value of TRUE means DMA is the default mode, and for cases where DMA cannot be used, the driver falls back to a non-DMA mode. A value of FALSE means non-DMA mode is the default and DMA mode will not be attempted.

For the driver to attempt to build the Scatter Gather DMA Buffer Descriptors, the upper layer should ensure that the buffer meets the following criteria.

- Start of the buffer should be a word aligned address.
- Number of bytes to transfer should be word aligned.

Due to cache coherency issues arising due to processor and SDMA access of the memory, the above criteria is further stringent for the read or receive operation (it is not applicable for write or transmit):

- Start of the buffer should be a cache line size (32 bytes) aligned address.
- Number of bytes to transfer should be cache line size (32 bytes) aligned.

27.4.4 Power Management

The primary methods for limiting power in SDHC module is to gate off all clocks to the controllers and to cut off power to the card slot when no cards are inserted. When a card is inserted to any of the slots, that slot alone is powered and the clocks to that controller alone are gated on. While using memory cards, the clock to the host controller and the clock to memory cards are gated off when ever the controller is idle. For SDIO cards, both the clocks stay on all the time.

SDHC driver supports the full power on and full power off states. In full power off state, the clocks to the controllers and the power to the inserted cards are turned off. When powered on, all cards inserted before and after the power down will be detected and mounted.

27.4.4.1 PowerUp

This function is implemented to support resuming a memory card operation that was previously terminated by calling PowerDown() API. Power to the card is restored, clocks to the pertaining controller is restarted.

SDHC driver is notified of a device status change. This results in signaling the SD bus driver of a card removal followed by a card insertion. The card is re-initialized and is mounted so that the all operations scheduled during a power down resumes. SDIO cards will be initialized on resume.

The details of this architecture and its operation can be found in the Platform Builder Help under the heading “Power On and Off Notifications for Secure Digital Card Drivers”, or in the online Microsoft documentation at the following URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddk5/html/wce50conpoweronoffnotificationsforsecuredigitalcarddrivers.asp

Note that this function is intended to be called only by the Power Manager.

27.4.4.2 PowerDown

This function has been implemented to support suspending all currently active SD operations just before the entire system enters the low power state. Note that this function is intended to be called only by the Power Manager. This function gates off all clocks to the controllers and powers down all the card slots.

27.4.4.3 IOCTL_POWER_CAPABILITIES

N/A

27.4.4.4 IOCTL_POWER_GET

N/A

27.4.4.5 IOCTL_POWER_SET

N/A

27.5 Unit Test

The SDHC driver is tested using the following tests included as part of the Windows CE 5.0 Test Kit (CETK).

- Storage Device Block Driver Read/Write Test
- Storage Device Block Driver API Test
- File System Driver Test
- Partition Driver Test
- Bluetooth Test
- Wi-Fi Test

27.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
SD Cards	SanDisk (128MB, 256MB, 512MB, 4GB) Kingston (256MB) NCP (128MB, 256MB, 512MB) Transcend (128MB, 256MB, 512MB) Toshiba (1GB)
MMC Cards	NCP (128MB) Transcend (128MB, 1GB)
SDIO Cards	SanDisk Connect Wi-Fi (SDWSDB-000) Socket Bluetooth (Toshiba SD-BT2)

27.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
rwtest.dll	Storage Device Block Driver Read/Write Test .dll file
disktest.dll	Storage Device Block Driver API Test .dll file
fsdtst.dll	File System Driver Test .dll file
msparttest.dll	Partition Driver Test .dll file
btw22.dll	Bluetooth Test .dll file (runs on the client CE device)
btwsvr22.exe	Bluetooth Test .exe file (runs on the server CE device)
wzctooltest.dll	Wi-Fi Test .dll file

27.5.3 Building the 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\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

27.5.4 Running the 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 Microsoft online documentation at the following URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcedebug5/html/wce50conCETKTests.asp

27.5.4.1 Storage Device Block Driver Read/Write Tests

Use the command line **tux –o –d rwtest –c "-z"** to run the tests. Note that this test tests only one card at a time.

27.5.4.2 Storage Device Block Driver API Tests

Use the command line **tux –o –d disktest –c "-z"** to run the tests. Note that this test tests only one card at a time.

27.5.4.3 File System Driver Test

Use command line **tux –o –d fsdtst –c "-p SDMemory –z"** to run the tests on an SD card. For MMC cards, use **tux –o –d fsdtst –c "-p MMC –z"**.

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

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

27.5.4.4 Bluetooth Test

The test procedures are well documented under "CETK Tests" in the Platform Builder Help and also available at the Microsoft online documentation.

Use two independent setups to test, where MX31 acts as the server in one and MX21 acts as the server in the other. To run the tests, both the server and the client boards should be connected to Ethernet and should have valid Ethernet IP addresses.

Run the Bluetooth server (btwsvr22.exe) on an MX21 board or MX31 board

Run the test (btw22.dll) on the client (MX31 board under test) using the command line **tux –o –d btw22 –c "<x.x.x.x>"** where x.x.x.x is the IP address of the server.

27.5.4.5 Wi-Fi Test

The Wi-Fi test suite requires 3 Wi-Fi access points to be present simultaneously, each configured for different encryption schemes. In case there is only 1 access point available, the tests can be split into 3 parts, depending upon encryption disabled, WEP 40-bit or WEP 104-bit respectively. Follow the steps below.

Create an ad-hoc Wi-Fi link (SSID: CE-ADHOC1) on a WLAN supported laptop or other WLAN-supporting device.

Copy tux.exe, kato.dll, tooltalk.dll and wzctooltest.dll to the Windows folder of the MX31 ADS board.

Insert an SDIO Wi-Fi card into the MX31 ADS SD/MMC slot of the controller to be tested.

Part I:

Setup the WLAN access point – SSID: CEOPEN. WEP: disabled

On the MX31 ADS board, run the command line **tux –o –d wzctooltest –x1-400,500,501**

Part II:

Setup the WLAN access point – SSID: CE-WEP40, WEP: 1/0x1234567890

On the MX31 ADS board, run the command line **tux –o –d wzctooltest –x411-413,502,503,410,451-452,450**

Part III:

Setup the WLAN access point – SSID: CEWEP104, WEP: 1/qwertyuiopasd

On the MX31 ADS board, run the command line **tux –o –d wzctooltest –x420**

Note that the access point should support complex encryption to run Part III. Also the Wi-Fi tests require wzctool.exe and ipconfig.exe, and hence the following catalog item needs to be included.

Catalog > Core OS > Windows CE Devices > Networking Features > Network Utilities

Note that before running each Part, system should be reset.

27.5.5 System Testing

The following system tests were 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.

The following system tests were performed to verify the operation of the SDIO Bluetooth.

- Use Start > Settings > Control Panel > Bluetooth Device Manager to scan for Bluetooth devices in the vicinity.

The following system tests were performed to verify the operation of the SDIO Wi-Fi.

- Connect to a Wireless Access Point, and be able to lease a valid IP address.
- Ping other Wi-Fi devices in the range.
- Connect to the Internet and browse.
- Stream audio and video files from Internet websites or from local intranet sites.
- Access media files shared on a remote PC and play those files.

27.6 Secure Digital Card Driver API Reference

Detailed reference information for the Secure Digital Card driver may be found in Platform Builder Help under the heading “Secure Digital Card Driver Reference”, or in the online Microsoft documentation at the following URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wceddk5/html/wce50lrfsecruredigitalcarddriverfunctions.asp

Chapter 28
Serial Driver

MX31 & MX32 has five internal UARTs (Universal Asynchronous Receiver Transmitters): UART1, UART2, UART3, UART4 and UART5 of which UART2 supports slow infrared communication and other UARTs support only serial communication. All the UART modules are capable of standard RS-232 non-return-to-zero (NRZ) encoding formats and UART2 supports slow infrared mode.

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 and the infrared I/O ports.

In the Windows CE 5.0 BSP implementation, the hardware-specific code that corresponds to the serial port driver's lower layer is implemented as the platform-dependent driver (PDD). This PDD will be linked with Microsoft provided public serial MDD library (com_mdd2.lib) to form the complete serial port driver.

28.1 Serial Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

28.1.1 For MX31, MX32 ADS

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\SERIAL
<TGTPLAT> CSP Driver Path	..\CSP\ARM\FREESCALE\<TGTSOC>\DRIVERS\SERIAL
CSP Static Library	<TGTSOC>_serial.lib, mxarm11_serial.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\SERIAL
Import Library	Com_mdd2.lib
Driver DLL	csp_serial.dll
Catalog Item for MX31 & MX32	Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −> Serial −> UART1 serial port Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −> Serial −> UART3 serial port Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −> Serial −>UART4 serial port Third Party −> BSPs −> Freescale <*TGTPLAT*> −> Device Drivers −> Serial −> UART5 serial port

Driver Attribute	Definition
SYSGEN Dependency	N/A
BSP Environment Variables for MX31 & MX32	BSP_SERIAL_UART1 =1 BSP_SERIAL_UART3 =1 BSP_SERIAL_UART4=1 BSP_SERIAL_UART5=1

28.2 Requirements

The serial port driver should meet the following requirements:

1. The driver shall support all the internal UARTs.
2. The driver shall support multiple serial ports.
3. The driver shall support two power management modes, full on and full off.

28.3 Hardware Operation

Refer to the chapter on the UART in the hardware specification document for detailed operation and programming information.

28.3.1 Conflicts with Other SoC Peripherals

28.3.1.1 MX31 & MX32 UART Peripheral Conflicts

MX31 and MX32 ADS has five internal UARTs: UART1, UART2, UART3, UART4 and UART5. In this UART1 and UART2 do not have conflicts with any other module and will be configured in functional mode. UART3 has conflicts with CSPI1 and CSPI3 modules and must be configured in alternate mode1. UART4 has conflicts with ATA and USB OTG modules and must be configured in alternate mode1. UART5 has conflicts with PCMCIA and USB modules and must be configured in alternate mode 2. The following table shows pins to be configured for serial driver for different UARTs.

UART Port	Pins To Be Configured	I/O MUX Settings
UART1	RXD1 TXD1 RTS1 CTS1 DTR_DCE1 DSR_DCE1 RI_DCE1 DCD_DCE1	Functional Mode
UART2	RXD2 TXD2 RTS2 CTS2	Functional Mode

UART Port	Pins To Be Configured	I/O MUX Settings
UART3	CSPI3_MOSI CSPI3_MISO CSPI3_SCLK CSPI3_SPI_RDY	Functional Mode
UART4	ATA_CS0 ATA_CS1 ATA_DIOR ATA_DIOW	Alternate Mode1
UART5	PC_VS2 PC_BVD1 PC_BVD2 PC_RST	Alternate Mode 2

MX31 and MX32 ADS board has three UARTs: UART A, UART B and UART C.

UART A can be mapped to UART1 or UART5 using the register setting on PBC. UART B can be mapped to UART3 or UART4 using the register setting on PBC. UART C can be mapped to UART1 or UART2 using the register setting on PBC. For more details, refer Peripheral Bus Controller document for MX31 ADS.

The current BSP supports the following COM ports. The interconnection between the internal UART and the DB-9 port on the MX31 and MX32 ADS for the given COM port is as described in the table below.

COM port	Internal UART/ port on MX31 & MX32 ADS
COM1	MX31/MX32's UART1 is connected to UART C on MX31/ MX32 ADS.
COM2	MX31/MX32's UART3 is connected to UART B on MX31/ MX32 ADS.
COM5	MX31/MX32's UART4 is connected to UART B on MX31/ MX32 ADS.
COM6	MX31/MX32's UART5 is connected to UART A on MX31/ MX32 ADS.

Notes:

1. This definition of COM ports is based on the current implementation in bspserial.c and platform.reg.

2. Since there is a conflict between UART3 and UART4 for connecting to the DB-9 port on MX31 & MX32 ADS, only one of COM2 and COM5 can be included at a time in the image.

In other words, we can include one set of UARTs from the following table at a time in image:

Set 1	Set 2
UART1(COM1) UART3(COM2) UART5(COM6)	UART1(COM1) UART4(COM5) UART5(COM6)

28.3.1.2 Active Sync Conflicts

Active Sync is only working the UART1 which is mapped with UART C of the board in MX31. Currently UART1 which is mapped to UART C of the board is configured as debug port. For active sync connection

we have to disable this debug port, which can be done by commenting the following line in the file WINCE500\PLATFORM\MX31\SRC\INC\ bsp_cfg.h

#define DEBUG_PORT DBG_UART1

ActiveSync is not supported in MX32 as the hardware does not support modem signals.

28.4 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 at the following location: **Developing a Device Driver –> Windows CE Drivers –> Serial Drivers –> Serial Driver Development Concepts**.

28.4.1 Power Management

Power management is not implemented in the serial driver.

28.4.2 MX31 Serial Registry Settings

The following registry keys are required to properly load the Serial driver.

```
; @CESYSGEN IF CE_MODULES_SERIAL
IF BSP_SERIAL_UART1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM1]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43F90000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:1
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM1\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="MX31 COM1 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_UART1

IF BSP_SERIAL_UART3
         IF BSP_SERIAL_UART4 !
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:5000C000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:2
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
   "Tsp"="Unimodem.dll"
```

```
   "DeviceType"=dword:0
   "FriendlyName"="MX31 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_UART4
ENDIF  ; BSP_SERIAL_UART3

IF BSP_SERIAL_UART4
        IF BSP_SERIAL_UART3 !
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM5]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43FB0000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:5
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM5\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="MX31 COM5 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_UART3
ENDIF  ; BSP_SERIAL_UART4

IF BSP_SERIAL_UART5
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM6]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43FB4000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:6
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM6\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="MX31 COM6 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_UART5
; @CESYSGEN ENDIF CE_MODULES_SERIAL
```

28.4.3 MX32 Serial Registry Settings

The following registry keys are required to properly load the Serial driver.

```
; @CESYSGEN IF CE_MODULES_SERIAL
IF BSP_SERIAL_UART1
; @XIPREGION IF PACKAGE_OEMDRIVERS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM1]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43F90000
   "IoLen"=dword:D4
   "Prefix"="COM"
```

```
   "Dll"="csp_serial.dll"
   "Index"=dword:1
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM1\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="mx32 COM1 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
; @XIPREGION ENDIF PACKAGE_OEMDRIVERS
ENDIF  ; BSP_SERIAL_UART1

IF BSP_SERIAL_UART3
    IF BSP_SERIAL_UART4 !
; @XIPREGION IF PACKAGE_OEMDRIVERS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:5000C000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:2
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="mx32 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
; @XIPREGION ENDIF PACKAGE_OEMDRIVERS
   ENDIF ; !BSP_SERIAL_UART4
ENDIF  ; BSP_SERIAL_UART3

IF BSP_SERIAL_UART4
    IF BSP_SERIAL_UART3 !
; @XIPREGION IF PACKAGE_OEMDRIVERS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM5]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43FB0000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:5
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM5\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="mx32 COM5 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
; @XIPREGION ENDIF PACKAGE_OEMDRIVERS
   ENDIF ; BSP_SERIAL_UART3
ENDIF  ; BSP_SERIAL_UART4

IF BSP_SERIAL_UART5
; @XIPREGION IF PACKAGE_OEMDRIVERS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM6]
```

```
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43FB4000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:6
   "Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM6\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="mx32 COM6 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
; @XIPREGION ENDIF PACKAGE_OEMDRIVERS
ENDIF  ; BSP_SERIAL_UART5
; @CESYSGEN ENDIF CE_MODULES_SERIAL
```

28.4.4 DMA Support

Serial driver uses the SDMA controller to transfer the data and this minimises the procesing that is required by the ARM core. Serial driver supports both DMA mode and Non-DMA mode of operation. We can enable/disable DMA using the boolean variable present in the file WINCE500\PLATFORM\<TGTPLAT>\SRC\INC\bsp_cfg.h.

Individual UARTs can be configured for DMA by using the variables and it is possible that some UARTs operate in one mode and the others in different mode (e.g., UART1 in DMA mode, UART3 in non-DMA mode).

To enable DMA, set the boolean variable to TRUE and for Non-DMA set the variable to FALSE. The following variables are used to enable/disable the DMA in MX31:

UART Port	MX31 & Mx32
UART1	BSP_SDMA_SUPPORT_UART1
UART2	Configured as SIR
UART3	BSP_SDMA_SUPPORT_UART3
UART4	BSP_SDMA_SUPPORT_UART4
UART5	BSP_SDMA_SUPPORT_UART5

When SDMA is enabled, buffers for Tx and Rx are allocated using HalAllocateCommonBuffer() in the initialization of the SIR driver. These buffers are used during the data transfer using SDMA.

DMA buffer size, both Rx and Tx, can be configured using the two variables defined in bsp_cfg.h. By default DMA buffer size is configured as

```
#define SERIAL_SDMA_RX_BUFFER_SIZE 0x200
#define SERIAL_SDMA_TX_BUFFER_SIZE 0x400
```

where SERIAL_SDMA_RX_BUFFER_SIZE is the receive DMA buffer size and SERIAL_SDMA_TX_BUFFER_SIZE is the transmit DMA buffer size.

28.5 Unit Test 1 (Serial Port Driver Tests)

The serial driver is tested using the Serial Port Driver Test and Serial Communications Test included as part of the Windows CE 5.0 Test Kit (CETK). The Serial Port Test assesses whether the driver supports configurable device parameters such as baud rate and data bits etc. The test also assesses additional functionality such as COM port events, escape functions and timeouts.

28.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
<*TGTSOC*> ADS board with serial port to be tested	Serial ports can be attached as COM1 through COMX.

28.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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

28.5.3 Building the Serial Port Driver 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\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

28.5.4 Running the Serial Port Driver Test

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.

Serial port tests are designed to test the serial port driver work properly and the API s behaves correctly, it should be pass all the test case.

The following table describes the Serial Port driver test cases and its descriptions.

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.

28.6 Unit Test 2 (Serial Communications Test)

The Serial Communications Test verifies the functionality of serial communications on a Microsoft Windows CE–based device. The Serial Communications Test evaluates basic functions, conducts stress tests, and tests hardware functionality. Before running the serial communication test cases make sure that the corresponding UART is not configured for other purposes like as debug UART.

The Serial Communications Test does not appear by default in the Windows CE Test Kit (CETK) window. You can add the test to the CETK with the User-Defined Test Wizard.

Serial Communications test is client server architecture and we have to start the server in one device before starting client on the second device.

28.6.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Serial null modem cable and available serial ports	Male-to-male null modem cable. A null modem cable differs from a standard serial cable in that it has its RX and TX lines crossed.
One <*TGTSOC*> ADS board with the serial port to be tested and another WINCE 5.0 device with a standard serial port.	Connect the serial port of the <*TGTSOC*> ADS board with the other WINCE 5.0 system's serial port using the null modem cable.

28.6.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
Pserial.dll	Test .dll file for Serial Communications Test

28.6.3 Building the Serial Communications Test

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

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

The Serial Communications Test does not appear by default in the Windows CE Test Kit (CETK) window. You can add the test to the CETK with the User-Defined Test Wizard.

28.6.4 Running the Serial Communications Test

The Serial Communications Test executes the ***tux –o –d pserial*** command line on default execution. You can modify the test by editing the command line. For information about how to edit the command line for a test, see Editing the Command Line for a Test. The following table shows the modifications you can make to the test.

To modify the Serial Communications Test, do the following:

Items	Add this command-line parameter
To specify a communications (COM) port where X is the number of the serial port	-p COMx
To run the test in master mode.	-m
To run the test in slave mode.	-s
To dump binary data	-d

Prior to running this test, confirm that all serial cable connections have a cable for debug output and a cable for serial communications. Verify that the RAS server does not run on your development workstation while the Serial Communications Test runs.

The Serial Communications Test requires two Windows CE–based devices. One Windows CE–based device must run as a client with the -s parameter while another Windows CE–based device runs as a server with the -m parameter. Each time you run this test, you must have one server and one client. The test does not support a server-to-server or client-to-client arrangement.

You should run the test on the target device with the target device being the client and also with the target device being the server. Tests that succeed when the target device is the server may fail when the target device is the client.

When you use the **-x** parameter to select individual test cases, always include test cases 10 and 11. Test case 10 verifies the serial connection and the ability of the two devices to synchronize. Test case 11 determines what properties the two serial ports share and is required when running the other test cases.

Note:- The test may abort if you run the test between a device running a desktop operating system (OS) and a Windows CE–based device. It is recommended that you use two Windows CE–based devices to run the test.

For detailed information on the Serial Port tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> Serial Communications Test –> Serial Communications Test Cases in the Platform Builder Help.

The following table describes the Serial Communication test cases and its descriptions.

Test	Description
10	Establishes communication between the Windows CE–based devices and synchronizes the devices.
11	Determines the set of serial port properties that the target device and development workstation have in common.
12 – 16	Checks serial port signals.
17	Checks the data read timeout options.
18	Checks the data write timeout options.
19	Checks serial port purging functions.

Test	Description
20	Evaluates XON/XOFF performance. This test case performs the following tasks: • Tests software flow control. If flow control does not work when the receiver sends an XOFF character, the sender rapidly transmits the rest of its data and the test fails. • Determines if the serial driver on the tested target device sleeps while waiting for data. • Determines whether the driver properly releases resources while sleeping. The test measures three IdleThreadDelta time values. The value of IdleThreadDelta determines how much time passes between consecutive OS calls to a low priority function. The larger the IdleThreadDelta value, the more work the OS performs and the less time the low priority function has to operate. The test measures the following three IdleThreadDelta values: • Sleep • Communications with software flow control disabled • Communications with software flow control enabled This test case fails if communication with software flow control enabled is more than 25 percent slower than communication with software flow control disabled.
21	Evaluates the reliability of the XON/XOFF protocol.
22	Evaluates the opening and closing of ports.
23	Tests communication with TX empty.
30	Tests roundtrip loopback time and verifies data at 9600 baud. The master sends out the data and the slave returns it.
31	Tests roundtrip loopback time and verifies data at 19200 baud. The master sends out the data and the slave returns it.
32	Tests roundtrip loopback time and verifies data at 38400 baud. The master sends out the data and the slave returns it.
33	Tests roundtrip loopback time and verifies data at 57600 baud. The master sends out the data and the slave returns it.
41	Tests the transmission speed of a 1-byte buffer with a fixed timeout value.
42	Tests the transmission speed of a 2-byte buffer with a fixed timeout value.
43	Tests the transmission speed of a 8-byte buffer with a fixed timeout value.
44	Tests the transmission speed of a 32-byte buffer with a fixed timeout value.
45	Tests the transmission speed of a 64-byte buffer with a fixed timeout value.
46	Tests the transmission speed of a 128-byte buffer with a fixed timeout value.
47	Tests the transmission speed of a 256-byte buffer with a fixed timeout value.
48	Tests the transmission speed of a 1024-byte buffer with a fixed timeout value.
51	Tests the reception speed of a 1-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
52	Tests the reception speed of a 2-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
53	Tests the reception speed of a 8-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.

Test	Description
54	Tests the reception speed of a 32-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
55	Tests the reception speed of a 64-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
56	Tests the reception speed of a 128-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
57	Tests the reception speed of a 512-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.
58	Tests the reception speed of a 1024-byte buffer with a fixed timeout value. If the **ReadFile** function times out before the read operation completes, the function returns the bytes that had been read when the function timed out.

28.7 Unit Test 3 (Serial Active Sync Test)

The serial active sync test verifies the synchronization of data between Windows CE based device and a desktop computer using serial driver. After synchronization with the device, we should be able to browse the device from desktop also able to transfer files between the device and the desktop computer. We have to install active sync application on the desktop computer. Active sync works with UART2, which supports all the signals in MX31.

28.7.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Serial cable	Serial cable is used to connect between device and the desktop computer.
Onei.MX31ADS board with the serial port to be tested and another desktop computer with a standard serial port.	Connect the serial port of thei.MX31ADS board with the desktop computers serial port using the serial cable.

28.7.2 Unit Test Software

Active sync application should be installed on the desktop computer.

28.7.3 Running Active Sync Test

The following are the steps involved in the active sync test:

1. Download the image to the MX31ADS board.
2. Select Start –> Settings –> Control Panel.
3. Open Network and Dial up Connection application.

4. Create a connection to the COM1 port:
5. Select Make a new connection.
6. Name the new connection as Test.
7. Select Direct connection Button.
8. Click Next button.
9. Select the device as MX31 COM1 UNIMODEM.
10. Click the configure Button
11. Set Flow control to none.
12. Keep baud rate as 19200.
13. Click OK Button.
14. Click Finish Button
15. Configure the board to use new connection as connection to other devices
16. Open the PC Connection Application from Control Panel directory

Make Sure that Enable direct connections to the Desktop Computer is checked.

17. Click the change connection button
18. Set the connection to “Test”
19. Click OK button
20. Click OK button

Connect the serial cable between the COM1 of the device and the desktop computer. Now active sync connection has to be established automatically and if this is not happens check the active sync application to make sure that it is enabled serial cable connection.

28.8 Serial Driver API Reference

Detailed reference information for the Serial driver may be found in Platform Builder Help at the following location:

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

28.8.1 Serial PDD Functions

The following table shows a mapping of Serial PDD functions to the functions used in the Serial driver:

PDD Function Pointer	Serial Driver Function
HWInit	SerSerialInit
HWPostInit	SerPostInit
HWDeinit	SerDeinit
HWOpen	SerOpen
HWClose	SerClose

PDD Function Pointer	Serial Driver Function
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

28.8.2 Serial Driver Macros

N\A

28.8.3 Serial Driver Structures

28.8.3.1 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;
```

Members

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 will be 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 timeout 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)

28.8.3.2 SER_INFO

This is a private structure contains the information about 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;
```

Members

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 will be read from registry.
dwIOBase	This static value contains the IO Base address of UART module which will be read from registry.
dwIOLen	This static value contains the IO length of UART Module which will be read from registry.

pBaseAddress	Pointer to the start address of the UART registers mapped.
cOpenCount	This value 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	This variables keep 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	This 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. (First use
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size

Chapter 29
Slow Infrared Driver

One of the i.MX31 internal UART i.e. UART2 supports both serial and slow infrared communication. In Slow Infrared (SIR) mode, UART2 provides infrared communication capability with external devices through use of external transceiver to provide low speed IrDA compatibility.

The slow infrared driver is implemented as part of the serial port driver. This implementation uses the standard stream interface and supports all the standard I/O control codes and entry points.

In the Windows CE 5.0 BSP implementation, the hardware-specific code that corresponds to the slow infrared driver's lower layer is implemented as part of the serial port driver’s platform-dependent driver (PDD). This PDD will be linked with Microsoft provided public serial MDD library (com_mdd2.lib) to form the complete serial port driver with slow infrared capability.

29.1 SIR Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

29.1.1 MX31 & MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\SERIAL
CSP Driver Path	..\CSP\ARM\FREESCALE\<TGTSOC>\DRIVERS\SERIAL
CSP Static Library	<TGTSOC>_serial.lib, mxarm11_serial.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\SERIAL
Import Library	Com_mdd2.lib
Driver DLL	csp_serial.dll
Catalog Item for MX31	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> Infrared Communication –> SIR
SYSGEN Dependency	SYSGEN_IRDA=1 SYSGEN_WINSOCK=1 SYSGEN_OBEX_INBOX=1
BSP Environment Variables for MX31	BSP_SIR_UART2=1

29.2 Requirements

The SIR driver should meet the following requirements:

1. The driver shall support the UART2 in slow infrared mode for MX31 & MX32
2. The driver shall support two power management modes, full on and full off.

29.3 Hardware Operation

Refer to the chapter on the UART in the hardware specification document for detailed operation and programming information.

29.3.1 The UART

The SIR driver interfaces with the Windows CE Serial Driver Architecture to provide the serial interface support

29.3.2 Conflicts with other SoC peripherals

29.3.2.1 MX31 & MX32 Peripheral Conflicts

UART2 has conflicts with FIR and configured in functional mode.

29.4 Software Operation

The SIR 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 at the following location: **Developing a Device Driver –> Windows CE Drivers –> Serial Drivers –> Serial Driver Development Concepts**.

29.4.1 Power Management

Power management is not implemented in the serial driver.

29.4.2 MX31 SIR Registry Settings

The following registry keys are required to properly load the SIR driver.

```
; @CESYSGEN IF CE_MODULES_IRDASTK && CE_MODULES_SERIAL
IF BSP_NOSIR !
#if (defined BSP_SIR_UART2 && !defined BSP_FIR)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIR]
   "DeviceArrayIndex"=dword:1
   "IoBase"=dword:43F94000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIR\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:6
   "FriendlyName"="MX31 SIR UNIMODEM"
   "DevConfig"=hex: 10,00, 00,00, 02,00,00,00, 10,01,00,00, 00,4B,00,00, 00,00, 08, 00, 00,
00,00,00,00
[HKEY_LOCAL_MACHINE\Comm\IrDA\Linkage]
    "Bind"=multi_sz:"Irsir1"
[HKEY_LOCAL_MACHINE\Comm\Irsir]
   "DisplayName"="MX31 SIR Driver"
   "Group"="NDIS"
   "ImagePath"="irsir.dll"
[HKEY_LOCAL_MACHINE\Comm\Irsir\Linkage]
   "Route"=multi_sz:"Irsir1"
[HKEY_LOCAL_MACHINE\Comm\Irsir1\Parms]
   "BusNumber"=dword:0
   "BusType"=dword:0
   "Port"=dword:4
   ; 0 = Use external dongle, 1 = Use internal IR
   "IntIR"=dword:0
   ; 0 - internal transceiver, 1 - Extended Systems JetEye dongle.
   "TransceiverType"=dword:1
```

```
   "DisablePowerManagement"=dword:1
#endif
ENDIF ;BSP_NOSIR !
; @CESYSGEN ENDIF CE_MODULES_IRDASTK && CE_MODULES_SERIAL
```

29.4.3 MX32 SIR Registry Settings

```
IF BSP_SIR_UART2
; @XIPREGION IF PACKAGE_OEMDRIVERS
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIR]
   "DeviceArrayIndex"=dword:1
   "IoBase"=dword:43F94000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIR\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:6
   "FriendlyName"="mx32 SIR UNIMODEM"
   "DevConfig"=hex: 10,00, 00,00, 02,00,00,00, 10,01,00,00, 00,4B,00,00, 00,00, 08, 00, 00,
00,00,00,00
[HKEY_LOCAL_MACHINE\Comm\IrDA\Linkage]
    "Bind"=multi_sz:"Irsir1"
[HKEY_LOCAL_MACHINE\Comm\Irsir]
   "DisplayName"="mx32 SIR Driver"
   "Group"="NDIS"
   "ImagePath"="irsir.dll"
[HKEY_LOCAL_MACHINE\Comm\Irsir\Linkage]
   "Route"=multi_sz:"Irsir1"
[HKEY_LOCAL_MACHINE\Comm\Irsir1\Parms]
   "BusNumber"=dword:0
   "BusType"=dword:0
   "Port"=dword:4
   ; 0 = Use external dongle, 1 = Use internal IR
   "IntIR"=dword:0
   ; 0 - internal transceiver, 1 - Extended Systems JetEye dongle.
   "TransceiverType"=dword:1
   "DisablePowerManagement"=dword:1
; @XIPREGION ENDIF PACKAGE_OEMDRIVERS
ENDIF ; BSP_SIR_UART2
```

29.4.4 DMA Support

SIR driver uses the SDMA controller to transfer the data and this minimises the procesing that is required by the ARM core. SIR driver supports both DMA mode and Non-DMA mode of operation. We can enable/disable DMA using the boolean variable present in the file WINCE500\PLATFORM\<TGTPLAT>\SRC\INC\bsp_cfg.h.

To enable DMA, set the boolean variable to TRUE and for Non-DMA set the variable to FALSE. The following variables are used to enable/disable the DMA in MX31:

MX31 & MX32
BSP_SDMA_SUPPORT_UART2

When SDMA is enabled, buffers for Tx and Rx are allocated using HalAllocateCommonBuffer() in the initialization of the SIR driver. These buffers are used during the data transfer using SDMA.

DMA buffer size, both Rx and Tx, can be configured using the two variables defined in bsp_cfg.h. By default DMA buffer size is configured as

```
#define SERIAL_SDMA_RX_BUFFER_SIZE 0x200
#define SERIAL_SDMA_TX_BUFFER_SIZE 0x400
```

where SERIAL_SDMA_RX_BUFFER_SIZE is the receive DMA buffer size and SERIAL_SDMA_TX_BUFFER_SIZE is the transmit DMA buffer size.

29.5 Unit Test1 (IR Port Tests)

The SIR driver is tested using the IR Port Test (Winsock 1.1 and Winsock 2.0). Test included as part of the Windows CE 5.0 Test Kit (CETK). These test cases are used to test the functionality of a driver for an IR Device using Winsock 1.1 and Winsock 2.0. These test cases are based on client server architecture and we need a pair of IR devices, also we have to start the server before starting the client.

29.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Two installed IR devices	You must install one IR device on a test server and another IR device on a test client. You should position the IR devices in close proximity to one another.

29.5.2 Unit Test Software

The following table lists the required software to run the unit tests.

Requirements	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
Irapi11.dll	Test .dll file for IR Port Test(Winsock 1.1)
Irapi22.dll	Test .dll file for IR Port Test(Winsock 2.0)
Irapisrv11.exe	Test .exe file for starting server for IR Port Test (Winsock 1.1)
Irapisrv22.exe	Test .exe file for starting server for IR Port Test (Winsock 2.0)

29.5.3 Building the IR Port Tests

29.5.3.1 IR Port Test (Winsock 1.1)

The IR Port Test (Winsock 1.1) come pre-built as part of the CETK. No steps are required to build these tests. The irapi11.dll and irapisrv11.exe can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

29.5.3.2 IR Port Test (Winsock 2.0)

The IR Port Test (Winsock 2.0) come pre-built as part of the CETK. No steps are required to build these tests. The irapi22.dll and irapisrv22.exe can be found alongside the other required CETK files in the following location:

[Drive]:\Program Files\Windows CE Platform Builder\5.00\cepb\wcetk\ddtk\armv4I

29.5.4 Running the SIR Tests

The following are the steps involved to run the IR Port test cases:

1. Download the image to the boards.
2. Select Start –> Settings –> Control Panel –> Network and Dial up Connections
3. Disable VMINI
4. Select Make a new connection from Network and Dial up connections
5. Select the Direct Connection Button
6. Click OK Button
7. Select the Device as <TGTSOC> SIR UNIMODEM
8. Click OK Button
9. Click Finish Button
10. Stop the obex services using the command “Services stop obx0:” on both the boards
11. Copy the server files irapisrv11 for Winsock 1.1 and irapisrv22 for Winsock 2.0
12. Copy the client files irapi11 for Winsock 1.1 and irapi22 for Winsock2.0
13. Keep the IR ports of both the boards closely.

Now boards are ready to run the IR Port test cases. We have to start Server in one board before starting the client in second board

29.5.4.1 IR Port Test (Winsock 1.1)

We can start the server in one device by typing ***irapisrv11*** at the command line. Then client side test executes on the second device using ***tux –o –d irapi11.dll*** in the command line.

For detailed information on the SIR Port tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> IR Port Test (Winsock 1.1) in the Platform Builder Help.

29.5.4.2 IR Port Test (Winsock 2.0)

We can start the server in one device by typing **irapisrv22** at the command line. Then client side test executes on the second device by using ***tux –o –d irapi22.dll*** in the command line.

For detailed information on the SIR Port tests, see Debugging and Testing –> Tools for Debugging and Testing –> Windows CE Test Kit –> CETK Tests –> IR Port Test (Winsock 2.0) in the Platform Builder Help.

The following table describes the IR Port Test (Winsock 1.1 and Winsock 2.0) cases and its description.

Test	Description
1 : Socket Create	Verifies that socket creation works for all valid parameters. This test case fails if creation of an IR socket fails.
2 : Socket Create Invalid	Verifies that socket creation fails for invalid parameters. This test case fails if creation of an IR socket succeeds with invalid parameters.
3 : Socket Close	Verifies that the **closesocket** function works for an IR socket with valid parameters. This test case fails if the **closesocket** function call fails.
4 : Com Port Usage	Verifies that when no IR sockets are in use, the Serial Infrared (SIR) driver does not use any communications (COM) ports. The test verifies that when an IR socket exists, any attempt to open a COM port that is attached to the SIR driver fails. This test case fails if a COM port can be opened while an IR socket uses the COM port.
5: Socket Memory Leak	Verifies that a call to the **socket** function and the **closesocket** function results in no loss of memory. The test checks the status of global memory before and after running the functions to determine if there is a memory leak. This test case always passes.
6 : Bind OK	Verifies that binding to a valid address succeeds. This test case fails if binding to a valid address fails.
7 : Bind Invalid Family	Verifies that binding to an invalid address family member returns an error. This test case fails if binding to an invalid address succeeds.
9 : Bind twice to the same socket	Attempts to bind to the same socket twice. The second bind is expected to fail. This test case fails if the first bind fails or the second bind succeeds.
10: Bind same name to 2 sockets	Attempts to bind to different sockets twice using the same name. The second bind is expected to fail. This test case fails if the first bind fails or the second bind succeeds.
11: Bind to a NULL socket	Attempts to bind to a NULL socket. This bind is expected to fail. This test case fails if the bind to the NULL socket succeeds.
12: Bind to a NULL address	Attempts to bind to a NULL address. This bind is expected to fail. This test case fails if the bind to the NULL address succeeds.
13: Bind large address	Attempts to bind the socket to various names of invalid length. This test case fails if the bind attempt succeeds.
14: Bind various addresses	Attempts to bind to various valid addresses. This test case fails if the bind attempt fails.
15: Bind getsockname	Verifies that the **getsockname** function works on a bound name. This test case fails if the **getsockname** function fails to retrieve names bound to the socket.

Test	Description
16: Bind, close, bind same name	Binds a name to a socket, closes the socket, and then binds the name to the socket to confirm that the name is again available for use. This test case fails if the bind attempt fails.
17: IrSIR open fail	Attempts to create an IR socket while the serial IR COM port is open. The attempt to create the socket should fail and the error should be WSAEBADF. This test case fails if the attempt to create the socket succeeds.
18: Bind memory leak	Verifies that calling varying binds does not result in memory loss. This test case always succeeds.
20: IRLMP_IAS_SET	Uses a simple test function that uses the IRLMP_IAS_SET socket option. This test case fails if the **setsockopt** function call fails with the IRLMP_IAS_SET socket option.
21: Invalid IRLMP_IAS_SET	Uses a simple set of tests that attempt to use an invalid attribute type with the IRLMP_IAS_SET socket option. This test case fails if the **setsockopt** function call succeeds with the IRLMP_IAS_SET socket option.
22: Integer IRLMP_IAS_SET	Attempts to use the **setsockopt** function for a variety of integer attributes. This test case fails if the **setsockopt** function call fails.
23: Octet sequence IRLMP_IAS_SET	Attempts to use the **setsockopt** function for valid and invalid octet sequences. This test case fails if the **setsockopt** function call fails in valid cases and succeeds in invalid cases.
24: User string IRLMP_IAS_SET	Attempts to use the **setsockopt** function for valid and invalid user strings. This test case fails if the **setsockopt** function call fails in valid cases and succeeds in invalid cases.
25: **ClassName** Len IRLMP_IAS_SET	Attempts to use the **setsockopt** function to set a class name. This test case fails if the **setsockopt** function call fails.
26: Attribute name Len IRLMP_IAS_SET	Attempts to use the **setsockopt** function to set an attribute name. This test case fails if the **setsockopt** function call fails.
27: Attribute Count IRLMP_IAS_SET	Determines whether an object has 256 or fewer attributes. This test case fails if more than 256 attributes are allowed.
28: IAS_SET memory leak	Verifies that calling various IAS_SET socket options results in no loss of memory. This test case always passes.
30: Bind hard-coded LSAP-SEL	Binds a variety of Logical Service Access Point Selector (LSAP-SEL) values. This test case fails if a bind attempt fails on a valid LSAP-SEL value or succeeds on an invalid LSAP-SEL value. This test case also fails if the Object Exchange Protocol (OBEX) server is enabled. The OBEX server reserves the port required by this test case.
31: Bind random LSAP-SEL	Binds null names to sockets, which causes the OS to assign a random LSAP-SEL value. This test case fails if the bind attempt fails.
32: Bind same hard coded LSAP-SEL	Binds a hard-coded LSAP-SEL value to two sockets and verifies that the WSAADDRINUSE error code is returned. This test case fails if the bind attempt succeeds.
42: Confirm integer IAS_SET	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribInt field on a client. The test verifies, from the remote server, that the value is set correctly. This test case fails if the value returned by the query from the remote server does not match the value that was set.
43: Confirm octseq IAS_SET	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribOctetSeq field on a client. The test verifies, from the remote server, that the value is set correctly. This test case fails if the value returned by the query from the remote server does not match the value that was set.
44: Confirm usrstr IAS_SET	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribUsrStr field on a client. The test verifies, from the remote server, that the value is set correctly. This test case fails if the value returned by the query from the remote server does not match the value that was set.

Test	Description
45: Confirm integer IAS_QUERY	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribInt field on the server. The test verifies, from the remote client, that the value is set correctly. This test case fails if the value returned by the query from the remote client does not match the value that was set.
46: Confirm integer IAS_QUERY	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribOctetSeq field on the server. The test verifies, from the remote client, that the value is set correctly. This test case fails if the value returned by the query from the remote client does not match the value that was set.
47: Confirm integer IAS_QUERY	Performs an IRLMP_IAS_SET socket option on a valid irdaAttribUsrStr field on the server. The test verifies, from the remote client, that the value is set correctly. This test case fails if the value returned by the query from the remote client does not match the value that was set.
48: Confirm attrib delete	Verifies that an Information Access Service (IAS) object is deleted when the socket for the object closes. This test case fails if the IAS object is not deleted when the socket for the object closes.
49: Confirm classNameLen query	Ensures that various class name lengths work correctly. This test case fails if the value returned by the remote query does not match the value that was set.
50: Confirm attribNameLen query	Ensures that various attribute names and lengths work correctly. This test case fails if the value returned by the remote query does not match the value that was set.
60: Bind and remote connect	Verifies that a bound socket can be connected to a remote device. This test case fails if the attempt to connect fails.
61: Bind all name lengths	Binds socket names of various lengths and attempts to connect using the remote server. This test case fails if the attempt to connect fails.
62: Bind all name values	Binds socket names of various lengths and attempts to connect using the remote server. The test attempts to use all legal character values in a name. This test case fails if the attempt to connect fails.
63: Bind sub and super strings	The server attempts to connect to the client using names that are substrings and superstrings of the bound name. If any connections occur, this test fails.
64: Bind/unbind then connect	Binds a name to a socket and then destroys the socket. The test then attempts to connect the remote side to the socket. This test case fails if the attempt to connect succeeds.
65: Connect-close before accept	Verifies that performing a **connect** function call and a **closesocket** function call prior to being accepted does not cause an exception. This test case fails if the **closesocket** function call fails.
66: Bind/connect hard coded LSAP-SEL	Attempts to bind the client using hard coded LSAP-SELs and then connect to the client from the server. This test case fails if the attempt to connect fails.
70: Connect to remote bind	Remotely binds a name to a socket on the server and then connects the client to the socket. This test case fails if the attempt to connect fails.
71: Connect all name lengths	The remote server binds socket names of various lengths and then the client attempts to connect. This test case fails if the attempt to connect fails.
72: Connect all name values	The remote server binds various socket names and then the client attempts to connect. This test case fails if the attempt to connect fails.
73: Connect sub and super strings	The client attempts to connect to the server using names that are substrings and superstrings of the bound name. If any connections occur, this test fails.
74: Connect to a bad device ID	Attempts to connect to a bad device identifier. The attempt to connect is expected to fail.
75: IrSir COM open fail then connect	Opens the COM port associated with the SIR driver, attempts to create an IR socket, closes the COM port, and then verifies that an IR socket can be created and connected to a remote device. The test then closes the IR socket and verifies that the COM port can be opened again. This test case fails if the COM port or IR socket cannot be opened and closed as expected.

Test	Description
76: Connect hard-coded LSAP-SEL	The server attempts to bind using hard-coded LSAP-SELs and the client attempts to connect. This test case fails if the attempt to connect fails.
78: Close connect in progress	Attempts to close a socket while simultaneously attempting to connect to a remote system. This test case fails if an exception occurs.
79: Connect twice with the same socket	Attempts to connect using a socket that has failed a prior attempt to connect. This test case fails if the second attempt to connect fails.
80: Enum devices disconnect and connect	Verifies that the enumeration of devices returns the same list while connected and disconnected. This test case fails if the two lists are not the same.
81: Enum devices while connected	Verifies that after a remote discovery operation followed by a connect operation, a local ENUM_DEVICES returns the remote address. This test case fails if the remote address is not discovered. This test cannot be run when the control connection uses an IR link. This test is intended to test the cache of device identifiers maintained by Infrared Link Access Protocol (IrLAP). You can run this test case once on a clean system, after which the cache is filled, making subsequent runs meaningless.

29.6 SIR Driver API Reference

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

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

29.6.1 SIR PDD Functions

The following table shows a mapping of SIR PDD functions to the functions used in the SIR driver:

PDD Function Pointer	SIR Driver Function
HWInit	SerIRInit
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

PDD Function Pointer	SIR Driver Function
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

29.6.2 SIR Driver Macros

N/A

29.6.3 SIR Driver Structures

29.6.3.1 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;
```

Members

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 charcters 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 will be the first parameter to the callback function.
dcb	This value contains the copy of Device Control Block.
CommTimeouts	This contains the copy of Commtimouts structure used to get and set the timeout 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)

29.6.3.2 SER_INFO

29.6.3.2.1 MX31 and MX32

This is a private structure contains the information about Serial port.

```
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;
```

Members

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 will be read from registry.
dwIOBase	This static value contains the IO Base address of UART module which will be read from registry.
dwIOLen	This static value contains the IO length of UART Module which will be read from registry.
pBaseAddress	Pointer to the start address of the UART registers mapped.
cOpenCount	This value 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	This variables keep 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	This 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. (First use
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size.

Chapter 30
Subscriber Identification Module (SIM) Driver

The Subscriber Identification Module (SIM) is a 32-bit peripheral used to communicate with the SIM cards or Eurochip pre-paid phone cards. It has two ports and hence can have direct control over two separate cards. However, the module cannot communicate with both the cards simultaneously.

The SIM driver transfers the data between SIM module and the SIM card. These data transfers follow the smartcard transmission protocol as defined in the ISO-7816 specification.

30.1 SIM Driver Summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\CSP\ARM\FREESCALE\MXARM11\DRIVERS\SIM
CSP Driver Path	N/A
CSP Static Library	mxarm11_sim.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\SIM
Import Library	Smclib.lib
Driver DLL	sim.dll
Catalog Item	ThirdParty −>BSPs −>Freescale<*TGTPLAT*> −>Device Drivers −>SmartCard−>SIM
SYSGEN Dependency	SYSGEN_SMARTCARD=1
BSP Environment Variables	BSP_SMARTCARD_SIM0=1 if Port0 is used BSP_SMARTCARD_SIM1=1 if Port1 is used

30.2 Requirements

The SIM driver should meet the following requirements:

1. The driver shall implement ISO-7816 protocol specification.
2. The driver shall support the SIM module for GSM SIM Cards.

30.3 Hardware Operation

Refer to the chapter on the SIM in the hardware specification document for detailed operation and programming information.

30.3.1 Conflicts with Other SoC Peripherals

30.3.1.1 MX31, MX32 Peripheral Conflicts

No conflicts.

30.4 Software Operation

The SIM driver uses the APIs of smartcard driver and follows the Microsoft-recommended architecture for smart card drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location: **Developing a Device Driver –> Windows CE Drivers –> Smart Card Drivers –> Smart Card Development Concepts**.

30.4.1 Power Management

The power management is currently not implemented for the SIM driver.

30.4.2 SIM Registry Settings

The following registry keys are required to properly load the SIM driver into the image.

```
IF BSP_SMARTCARD_SIM0
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIM]
    "Prefix"="SCR"
    "Dll"="sim.dll"
    "Index"=dword:1
    "Order"=dword:11
    "IOBase"=dword:50018000
    "AlternateIOBase"=dword:0
ENDIF   ;BSP_SMARTCARD_SIM0

IF BSP_SMARTCARD_SIM1
IF BSP_SMARTCARD_SIM0!
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIM]
    "Prefix"="SCR"
    "Dll"="sim.dll"
    "Index"=dword:2
    "Order"=dword:11
    "IOBase"=dword:50018000
    "AlternateIOBase"=dword:0
ENDIF;  BSP_SMARTCARD_SIM0!
ENDIF;  BSP_SMARTCARD_SIM1
```

SIM driver is just a variance of the smartcard driver & uses the smartcard layer provided by Microsoft as its upper layer. Thus, in order to use the SIM driver, it is required to first include the smartcard layer into the OS design and then build the image.

30.5 Unit Test

There is no Windows CE 5.0 Test Kit (CETK) available to test the SIM driver. However, the driver could be tested by running the SIM Application present in the WINCE500\SUPPORTS\APPS folder.

30.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
GSM SIM card	Any GSM SIM card.

30.5.2 Building the SIM Application

The SIM Application is built using the eMbedded Visual C++ tool provided along with Windows CE Kit. Create a new workspace for this SIM application and build it using the eMbedded Visual C++ tool.

Once the application is built, copy the SIMApp.exe file along with its supporting files into the *\WINCE500\PBWorkspaces\MX31Mobility\RelDir\MX31_ARMV4I_Debug* directory if working in Debug mode or into the *\WINCE500\PBWorkspaces\MX31Mobility\RelDir\MX31_ARMV4I_Release* directory if working in the Release mode.

30.5.3 Running the SIM Application

Within Platform Builder, go to the **Target** menu option and select the **Run Programs** menu option. This will give a list of applications that could be run on the OS. Select SIMApp.exe from this list and click on Run to run this application.

Once the SIM application is loaded, we get a screen on the display containing three drop down menus: **Tools**, **About** and **File** in the tool bar. By now, the SIM application would have sent 3 SELECT_FILE commands to the SIM card to open the Master File (MF), Dedicated File (DFTELECOM) which contains telecom services features and an Elementary File (EF). Once the EF is opened, the application can issue commands to parse through the records in this file.

The user can choose the **Tools** option from the toolbar and then choose the **Phonebook** option. Here the user can parse through the phonebook entry, add a new entry into the phonebook or remove an existing entry.

The user can move to the previous record by clicking on the Previous button, the next record by clicking on the Next button. If there does not exist any data in a selected record, the application will prompt a message referring to an empty record.

In order to add/modify an entry into the SIM, it is first required to edit the entry and then click on the Modify button which will save the modifications.

The following are the functions present in the application that are called when user performs any operation in the **Phonebook** option in the GUI.

Functions	Description
SIMParsePBRecord	This function is used to parse the phone book data from the record.
SIMAssemblePBRecord	This function assembles the phone book record.
SIMDataExchange	This function is used to exchange data (apdu) between the card reader and the SIM card using T0 protocol.
SIMStatus	This function gets currently selected file's status, which specifies the allowed operation on that file
SIMSelectFile	This function uses the SELECT_FILE command to select a file present in the SIM card.
SIMReadRecord	This function uses the READ_BINARY command to read a string of bytes from a selected file.
SIMUpdateRecord	This function uses the UPDATE_RECORD command to update one complete record in the selected file.

30.6 SIM Driver API Reference

Detailed reference information for the SIM driver API may be found in Platform Builder Help at the following location:

Developing a Device Driver –> Windows CE Drivers –> Smartcard Drivers –> Smartcard Driver Reference.

30.6.1 SIM PDD Functions

The SIM driver is developed on the lines of the sample PC Card smart card readers namely Bulltlp3.dll, Pscr.dll, and Stcusb.dll that are included in the Software Development Kit (SDK).

30.6.2 SIM Driver Macros

N/A

30.6.3 SIM Driver Structures

30.6.3.1 ClockRateFactor

This structure is used to store the Clock Rate Conversion Factor table defined in ISO-7816 specification.

```
typedef struct {
  UINT8 FI;
  INT16 Fi;
  INT16 f;
} ClockRateFactor;
```

Members

FI

Bit form representation of Clock Rate Conversion Factor as received from SIM Answer To Reset (ATR)

Fi

Clock Rate Conversion Factor

f

Maximum clock frequency in MHz

30.6.3.2 ClockRateFactor

```
typedef struct {
    UINT8   DI;
    UINT8   Di;
}BaudRateFactor;
```

Members

DI

Bit form representation of Baud Rate Adjustment Factor as received from SIM ATR (Answer To Reset)

Di

Baud Rate Adjustment Factor

Chapter 31 Touch Panel Driver

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

31.1 Touch Panel Driver summary

The following table provides a summary of source code location, library dependencies and other BSP information:

For MX31, MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
MXARM11 CSP Driver Path	..\ CSP\ARM\FREESCALE\MXARM11\DRIVERS\TOUCH
CSP Driver Path	N/A
CSP Static Library	mxarm11_touch.lib
Platform Driver Path	..\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\TOUCH
Import Library	N/A
Driver DLL	touch.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*> –> Device Drivers –> MBX–> MC13783 Touch Driver
SYSGEN Dependency	SYSGEN_Touch = 1
BSP Environment Variables	BSP_PMIC_MC13783=1,BSP_TOUCH_MC13783=1

31.2 Requirements

The touch panel should conform to the standards as explained in PB Documentation under

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

31.3 Hardware Operations

The hardware consists of a LCD Panel. For proper functioning we need ADC module, used to generate the touch samples which after proper calculation can be converted to the x,y coordinates.The ADC module and the Touch Interrupt are part of PMIC. The more details about them can be found in Chapter 26.12..

31.4 Conflicts with SOC peripherals

For iMX31 and iMX32, the conflicts are only with the GPIO Pin with which the PMIC Interrupt is routed. So those pins cannot be used as standard GPIO. Please refer Chapter 26 for PMIC Interrupt routing and conflicts.

31.4.1 Conflicts with iMX31, iMX32

The Touch Driver needs a Timer to provide the necessary timings between different ADC Samples. So EPIT2 is dedicated for Touch Panel and hence can not be used by any other module.

31.5 Software Operations

The touch screen driver reads user input from touch screen hardware and converts it to touch events that are sent to the Graphics, Windowing, and Events Subsystem (GWES). The driver also converts uncalibrated 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 while the user's finger or stylus is touching the touch screen. When the user's finger or stylus is removed from the screen, the driver must submit at least one final event indicating that the user's 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 cross hair 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.

31.5.1 Touch driver registry settings

```
IF BSP_NOTOUCH !

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="touch.dll"
    "MaxCalError"=dword:10

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

```
    "DblTapTime"=dword:637

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "MaxCalError"=dword:7
    "CalibrationData"="539,520 280,259 280,778 793,781 794,259"

; For TouchPanel calibration. Note that the Windows Mobile PocketPC touchpanel
; calibration is handled automatically by the welcome.exe application so a
; separate "Launch" registry key is not required. Also, Windows Mobile
; SmartPhone does not support a touchpanel at all which means that this is
; not required for SmartPhone either.
#if !defined IMGTPC && !defined IMGPPC
[HKEY_LOCAL_MACHINE\init]
    "Launch80"="touchc.exe"
    "Depend80"=hex:14,00, 1e,00
#endif ; !defined IMGTPC && !defined IMGPPC

ENDIF ; BSP_NOTOUCH !
```

31.6 Unit Tests

31.6.1 Unit test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
SHARP LQ035Q7DB02 QVGA Panel	Display panel required for display of graphics data.

31.6.2 Unit test software

The following table lists the required software to run the unit tests.

Requirements	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
Touchtest.dll	The Test .dll File
Touch.dll	Touch Panel Driver

31.6.3 Building the touch panel Tests

The touch test cases can be run by typing :

tux -o -d touchtest.dll -x <Test case id>.

The test case ids are described in PB Documentation:

Debugging and Testing > Tools for Debugging and Testing > Windows CE Test Kit > CETK Tests > Touch Panel Test

Test case no 8007 is skipped since the required API TouchPanelInitializeCursor is not implemented in the driver.

31.7 Touch Panel API reference

The complete API reference is given in PB documentation at :

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

Chapter 32
USB OTG Driver

The OTG USB driver provides High Speed USB 2.0 host and device support for the USB “On The Go” (OTG) port of the MX31/MX32. The OTG driver will automatically select either host or device functionality for high speed (in case of MX31/MX32) at any given time, depending on the USB cable/mini-plug configuration. This is achieved by the set of three drivers: USB OTG host controller driver, USB client driver and/or USB transceiver controller (“Full Function”) driver, which performs the host/function client switching.

The USB host driver can be configured for class support for mass storage, HID, printer, and RNDIS peripherals. The device/client portion can be configured to provide one of mass storage, serial, or RNDIS function.

The “Full Function” OTG transceiver driver automatically selects between the host or client driver. The host or client can also be configured as the only mode for the OTG port, using the “Pure Host” or “Pure Client” catalog item. All the OTG catalog items are exclusive. (See summary sections below).

32.1 USB OTG Driver Summary

32.1.1 OTG Client Driver Summary

32.1.1.1 For MX31/MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31,MX32ADS
Target SOC (TGTSOC)	MX31,MX32
CSP Driver Path	PUBLIC*\CSP\ARM\Freescale\<*TGTSOC*>\Drivers\USBD
CSP Static Library	mvxarm11_usbfn.lib
Platform Driver Path	\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\USBD
Import Library	<*TGTSOC*>_ufnmddbase.lib
Driver DLL	usbfn.dll

Driver Attribute	Definition
Catalog Item	High Speed OTG: Third Party —> BSPs —> Freescale <*TGTPLAT*>: ARMV4I —> Device Drivers —> USB Devices —> USB High Speed OTG Device To support only client/device mode, choose .. —> High Speed OTG Port Pure Client Function When ..—> High Speed OTG Port Full OTG Function is selected, the client driver will also be included in the build along with the transceiver and host.
SYSGEN Dependency	SYSGEN_USBFN
BSP Environment Variable	BSP_USB BSP_USB_HSOTG_CLIENT

Note that USB clients require a function driver to be loaded. A client can only present one function. Only one of the function drivers (described in section 37.4.5) should be configured through drag and drop. If more than one is configured, the Serial function will be default unless the registry is manually modified.

32.1.2 OTG Host Driver Summary

32.1.2.1 For MX31/MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	PUBLIC*\CSP\ARM\Freescale\<*TGTSOC*>\Drivers\USBXVR PUBLIC*\CSP\ARM\Freescale\<*TGTSOC*>\Drivers\USBH\EHCI
CSP Static Library	N/A
Platform Driver Path	\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\USBH\HSOTG
Import Library	<TGTSOC>_ehci_lib.lib <TGTSOC>_Ehcdmdd.lib <TGTSOC>_hcd2lib.lib
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party —> BSPs —> Freescale <*TGTPLAT*>: ARMV4I —> Device Drivers —> USB Devices —> USB High Speed OTG Device To support only host mode, choose .. —>High Speed OTG Port Pure Host Function When ..—>High Speed OTG Port Full OTG Function is selected, the host driver will also be included in the build along with the transceiver and client.
SYSGEN Dependency	SYSGEN_USB
BSP Environment Variable	BSP_USB BSP_USB_HSOTG_HOST

Note: Host driver requires a set of class drivers to be loaded. See High Speed Host H2 (37.5) for class driver information.

32.1.3 OTG Transceiver Driver Summary (For HIGH-SPEED only)

32.1.3.1 For MX31/MX32

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	PUBLIC*\CSP\ARM\Freescale\<*TGTPLAT*>\Drivers\USBXVR
CSP Static Library	<TGTSOC>_xvc.lib
Platform Driver Path	PLATFORM\SRC\DRIVERS\USBXVR
Import Library	N/A
Driver DLL	imx_xvc.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*>: ARMV4I –> Device Drivers –> USB Devices –> USB High Speed OTG Device –> High Speed OTG Port Full OTG Function
SYSGEN Dependency	SYSGEN_USBFN
BSP Environment Variable	BSP_USB BSP_USB_HSOTG_CLIENT BSP_USB_HSOTG_HOST BSP_USB_HSOTG_XVC

32.2 Requirements

1. The High Speed OTG driver must support USB specification 2.0 with OTG extension for host/client switching.
2. The driver shall be configured as client/peripheral by default, with one function driver defined as default. When nothing is connected to the OTG port, the port shall not drive Vbus and shall await attachment to a host by raising its D+ signal. On attachment of a mini-A plug the driver shall switch to host mode as described below.
3. When a mini-B plug is connected to the OTG port, and the cable's opposite end is connected via mini-A plug to another OTG device, or via A-type plug to a host, then the OTG shall initiate operation as peripheral and respond to USB protocol from the host.
4. When a mini-A plug is connected to the OTG port and the cable's opposite end is connected via mini-B plug to another OTG device, then the OTG shall initialize/re-initialize itself into host mode and begin to act as a host. The OTG port remains in host mode whenever a mini-A plug is mated to the OTG socket connector.
5. The OTG port as client/peripheral shall support mass storage, RNDIS and serial clients
6. The OTG port as host shall support mass storage, printer, HID and RNDIS classes
7. When nothing is attached to the OTG port, the driver shall configure the controller and transceiver into a low power state.
8. When the system is suspended with nothing attached to the OTG port, the system shall wake upon attachment of the port to a host or attachment of a device with mini-A plug.

9. When the system is suspended while the OTG port is connected to a host or to a device with a mini-A plug, the system shall remain suspended when the OTG port connection is unplugged.
10. When the system resumes after suspend, any attached devices shall be enumerated and their class drivers loaded appropriately.
11. Data transfer rates on the client port shall exceed 40 Mbits/sec in Mass Storage client.

32.3 Hardware Operation

There are two physical OTG mini sockets on the ADS, one for USB High Speed support, and one for USB Full Speed OTG, connected to appropriate USB transceivers. The <*TGTSOC*> contains a USB 2.0 core for handling OTG, which is connected to the external transceivers via the <*TGTSOC*> IO multiplexer IOMUX). External buffer components on the ADS select the signal from appropriate pins for correct Controller–>transceiver connection. Other options are possible on <*TGTSOC*> hardware (such as routing signals on H1 signal port to OTG transceiver, with both controllers “offline”). This simply allows external peripherals direct connection to a transceiver. The OTG driver currently supports only one hardware MUX configuration.

The ID Pin Detect is supported via the Transceiver Driver (for High Speed OTG), and is constructed as a stream interface driver.

Please go through the sample reference implementation that is provided with WinCE 5.0 installation first to get more detail understanding on how USB Host controller and USB Function controller driver are structured.

The <*TGTSOC*> processor supports speed translation within the USB 2.0 controller, a non-standard EHCI implementation. As a result software does not currently support full/low speed devices (aside from those non-FS hub devices directly connected to the OTG port).

The High Speed OTG port can be used in combination with High Speed and Full Speed USB host ports.

The <*TGTSOC*> ADS can supply a total of 100mA to attached devices on the OTG port and the default behavior does not need to be modified.

All bus powered hubs that have been tested require 500mA and therefore are not supported for use with the ADS. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100mA (For Ex : Mini HDD devices should be connected through self powered Hub).

32.3.1 Conflicts with other Peripherals

32.3.1.1 MX31/MX32 OTG Peripheral Conflicts

The High Speed OTG port conflicts with UART4. The USB controller drivers coordinate in their management of the USB peripheral block clock and processor core voltage, as described in section 37.4.4.1 and 37.4.4.2.

32.4 Software Operation

32.4.1 USB OTG Host Controller Driver

This driver enables the USB host functionality for the OTG port. It is part of the standard Windows USB software architecture as shown in Figure 6.

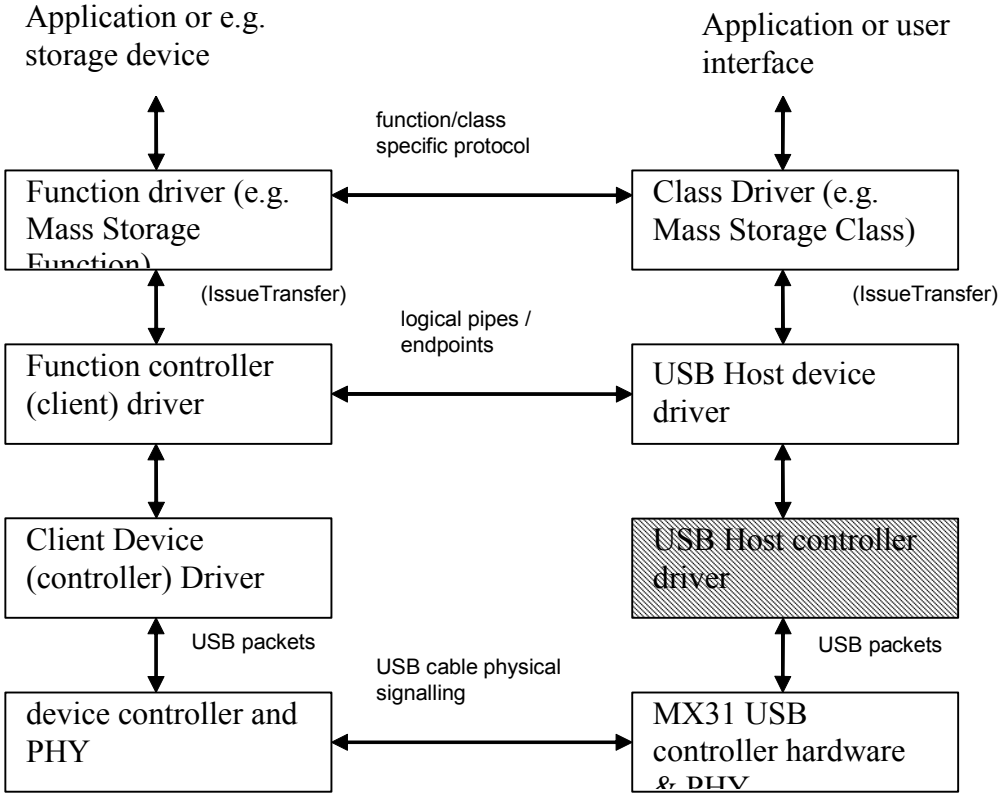


Figure 6: Windows USB Driver Architecture

For further details of the Windows CE USB driver architecture and usage, see Platform Builder Windows CE 5.0 help topic:

Developing a Device Driver –> Windows CE Drivers –> USB Host Drivers

and

Developing a Device Driver –> Windows CE Drivers –> USB Host Drivers –> USB Host Controller Drivers –> USB Host Controller Driver Development Concepts

When transceiver mode is included, the host driver is activated when a USB Mini-A plug is connected to the Mini USB OTG socket. When Pure Host mode only is selected, the host driver is always in control of the relevant USB controller.

When a USB device is connected to the Mini USB OTG socket of the ADS, the host controller driver enumerates and activates the appropriate class driver (see 37.4.1).

Windows CE 5.0 supports the following USB class drivers:

- Mass Storage -- SD cards, MMC cards, CF cards, HDD drive, thumb drive (disk-on-key).
- Note: some card reader firmware is not supported by the Microsoft standard Mass Storage class driver.
- HID -- Keyboard and mouse
- Printer
- RNDIS – Network Device Interface communication class

Hubs are supported, in all configurations with Full and Low Speed peripherals.

This version of the host controller driver has been verified against the following USB 2.0 vendor devices:

- Various disk-on-key including Kingston and Memorex
- HP HS self-powered and bus-powered hub
- External self-powered HS hard drive
- HS and FS card reader with CF and MMC storage. (Note: there are known issues formatting large CF cards some FS card readers)
- HP Photosmart 7450 printer and Lexmark E232 Laser Printer
- A variety of cameras, including Sony Cybershot and HP717
- Logitech keyboard and mouse

32.4.1.1 User Interface

As described above, user access to the USB host driver is via class drivers. For further details on these Host Client Drivers refer to Windows CE 5.0 Platform Builder help topic:

Developing a Device Driver –> Windows CE Drivers –> USB Host Drivers –> USB Host Controller Drivers –> USB Host Client Drivers.

Where new class driver code is to be developed, refer to the Host client driver interface functions (e.g. IssueBulkTransfer) as documented in:

Developing a Device Driver –> Windows CE Drivers –> USB Host Drivers –> USB Host Controller Drivers –> USB Host Client Drivers –> Host Client Driver Reference.

32.4.1.2 Host Controller Configuration

The driver is configured into the BSP build by dragging & dropping the appropriate catalog item for USB HS OTG. By default, host support is included along with peripheral/device and transceiver support. Additional classes to be supported must also be selected from the Core OS catalog. See Registry Setting OTGSupport (below) for details on excluding OTG host support from the build.

The internal <*TGTSOC*> USB OTG signals can be multiplexed to a choice of pins on IC, as described for the IOMUX in the hardware reference manual. The ADS also supports multiplexing these OTG signals through a choice of buffers in one of two configurations.

The USB OTG driver is currently supports one specific hardware configuration, which defines the other on-chip peripherals and devices with which an OTG device can be simultaneously configured in the BSP.

32.4.1.3 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 at driver initialization. Unless physical addresses are specified in API accesses at the class-driver interface, the driver copies data between the user/class-provided data buffers and the DMA buffer from the driver physical memory pool.

The default DMA physical memory pool size is 128 kB. This value can be altered by registry setting PhysicalPageSize.

32.4.1.4 Vbus/Configured Power

USB provides a means to monitor the configured power of devices attached to a USB host. The host driver verifies that each attached device will not exceed the configured power limit.

This power limit is implemented via the platform-specific function `BSPUsbCheckConfigPower()` as described in section 37.4.1.8.1 and located in:

`PLATFORM\<TGTPLAT>\SRC\Drivers\USBH\Common\hwinit.c`

This function must be modified to correspond with the platform hardware capabilities.

The <*TGTSOC*>/ADS can supply a total of 100mA to attached devices on the OTG port and the default behavior does not need to be modified.

All bus powered hubs that have been tested require 500mA and therefore are not supported for use with the ADS. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100mA.

32.4.1.5 Registry Settings

The USB OTG host controller settings are values located under the registry key:

`[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSOTG]`

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization.

Default values are contained in hsotg.reg.

Value	Type	Content	Description
Dll	sz	hcd_hsotg.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable host driver on the OTG. If no host support is required (client only) then this value can be set to 0, though the HCD_HSOTG key is not normally configured in the image at all when pure Host function is selected.

Value	Type	Content	Description
OTGGroup	sz	01	This unique string (example “00” to “99”) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance. Only one instance of the OTG is actually supported currently on the *<TGTSOC>* hardware.
HcdCapability	dword	4	HCD_SUSPEND_ON_REQUEST. Note: HCD_SUSPEND_RESUME is always assumed.
PhysicalPageSize	dword	20000	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 will be at least 3x as large as the largest data buffer transferred using IssueTransfer().

32.4.1.6 Host USB Test Modes

The USB 2.0 specification defines PHY-level test modes for USB host ports (see definitions in USB 2.0 specification section 7.1.20).

The MX31/MX32 USB host drivers support “Packet” test mode. The test mode is configured by compiling the BSP with the compilation flag OTG_TEST_MODE defined within bsp_cfg.h:

```
#define OTG_TEST_MODE
```

This configures the appropriate host controller within the platform-specific hardware initialization function (ConfigOTG()), located in:

For MX31:

```
PLATFORM\MX31\SRC\Drivers\USBH\Common\hwinit.c
```

For MX32:

```
PLATFORM\MX32\SRC\Drivers\USBH\Common\hwinit.c
```

The test mode is entered upon initialization, and can not be exited. Normal USB operation is disabled when test mode support is compiled into the image.

32.4.1.7 Unit Test

The USB driver has many devices to be tested. Tests are performed manually and include connecting the devices, and confirming the attach, detach (on unplug) re-attach (on subsequent plug in of device), and transferring and verifying data (and/or functions).

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.

32.4.1.7.1 USB Host Controller Driver Test

The information in this section is (c) 2001 Microsoft Corporation, and is documentation for the Windows CE 5.0 CETK USB Host tests.

This should normally be found under Platform Builder / Windows CE product documentation:

Debugging and Testing –> Windows CE Test Kit –> CE Test Kit

32.4.1.7.2 Steps to build the image to be tested

1. Checkout the RTM to be tested or install the MSI provided.

2. Add the following components from the catalog

-- Freescale <*TGTPLAT*> :ARMV4I-Device Drivers-USB Devices - Add the USB item that needs to be tested

-- 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 etc.)

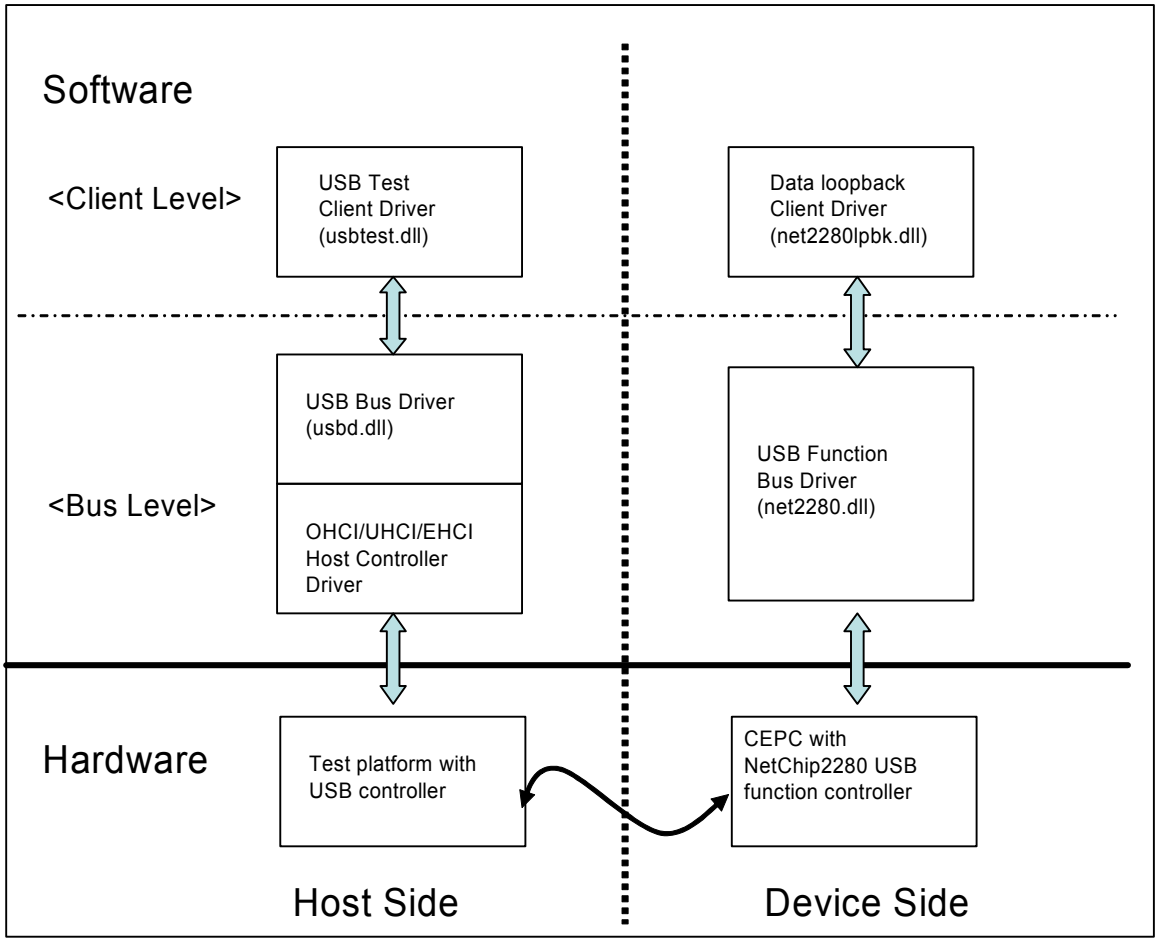
-- Core OS - Windows CE devices - File Systems And Data store- Storage Manager; (Sub-Components: FAT File System, Partition Driver, Storage Manager control panel applet)

-- Device Drivers- USB Function-USB Function Clients-Serial.

32.4.1.7.3 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, please see 37.4.1.1), also it can be used to verify whether a certain USB host controller (either stand alone card or onboard logic) can work with Windows CE or not.

The test setup and scenario is shown in the following picture:



This test suite acts as a client driver above USB bus driver (usbd.dll), it will be loaded when test device is connect to host through USB cable. Here, test device is a CEPC with a NetChip2280 USB function controller card in it. After this CEPC is booted up and net2280lpbk.dll got loaded, the whole CEPC will act as a generic USB data loopback device. USB test suite (the test client driver on the host side) can then streaming data or issue device requests to and from this data loopback device; this is how the USB host controller and its corresponding host controller drivers got exercised.

NetChip2280 USB function PCI controller card is a USB2.0 compatible USB function device, so we can use it to test both USB2.0 and USB1.1 host controllers (EHCI/OHCI/UHCI) and corresponding drivers.

Netchip2280 controller has 6 endpoints besides endpoint 0. The data loopback driver (net2280lpback.dll) configures these endpoints to be three pairs: one bulk IN/OUT pair, one Interrupt IN/OUT pair, and one Isochronous IN/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.

32.4.1.7.4 Hardware Requirements

- Test platform
- Host Controller Card (if not onboard logic)

- CEPC
- Netchip2280 Card
- USB cable

32.4.1.7.5 Required Software

Host side:

- Tux.exe
- Ddlx.dll
- Usbtest.dll
- Tooltalk.dll
- Kato.dll
- USB component (usbd.dll, EHCI/OHCI/UHCI host controller driver(s)) must be included in the run time image.

Device side:

- Lufldrv.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image.

32.4.1.7.6 Running the Test

The test procedure is list as following:

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 there is NO connection between host side and device through USB cable. Then launch command s tux –o –d ddlx –c “usbtest” “–xYYYY”, “YYYY” is the test case(s) you want to run.
5. The test will first remind you to make sure there is NO connection between host and device side. Then after 7 seconds, the test will ask you to connect two sides with USB cable.
6. After you connected two sides with USB cable, the test main body will start to run.
7. After test(s) is(are) done, and if you want to run some other tests in the test suite, you do not need to disconnect two sides, just type the test command, and the tests will start directly. Of course, if you disconnected the USB connection before the second run, tests will ask you to connect again before start the real test part.

32.4.1.7.7 Test Cases

The test suite contains following test cases:

Test Case ID	1001-1315, 1501-1515
Test Case ID	1001-1315, 1501-1515
Test Description	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) *. Concerning about how data is being transferred, there are also five categories: 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 we exercise both synchronous and asynchronous transfer methods (test cases like xx1x using asynchronous transfer method, test cases like xx0x using synchronous method. So totally there are 5*5*2 = 50 test cases.

Test Case ID	1401-1413
Test Description	Some additional data loopback tests. They mainly focus on testing APIs like GetTransferStatus(), AbortTransfer() and CloseTransfer().

Test Case ID	2001-2013
Test Description	Test related with Device requests.

Test Case ID	9001-9004
Test Description	These are some special tests that test APIs like SuspendDevice(), ResumeDevice() and DisableDevice().

Test Case ID	9005
Test Description	This is a test that stresses EP0 transfer (Vendor Transfer)

* 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; they are DWORD aligned, neither too big nor too small. By having all tests list above passed under this

configuration is more than good enough 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, this will be a good enrichment for testing.

This test suite has provided a way to change packet sizes of on NetChip2280 device on the fly. They are:

1. Test case 3001: Using some very small packet sizes in NetChip2280 device's full speed configuration.
2. Test case 3002: Using some very small packet sizes in NetChip2280 device's high speed configuration.
3. Test case 3003: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device's full speed configuration.
4. Test case 3004: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device's high speed configuration.
5. Test case 3005 (High Speed only): Using some very large packet sizes (like 2*1024 for Isochronous endpoints) in NetChip2280 device's full speed configuration. (Note: In the real world, Netchip2280 can not 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, through PB you should see usbtest.dll being unloaded and loaded again automatically. It means the packets sizes on netchip2280 side have already been changed. Then you can run those normal tests. You can use test case 3011(for full speed config) and 3012 (for high speed) to restore default packet sizes.

Another category test that is important for USB2.0 host controllers and drivers is called golden bridge tests, which means USB2.0 host controller is connected with a full speed (USB1.1) device. This is the only scenario that USB2.0 host controller will perform split transfers.

We can force NetChip2280 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 will force Netchip2280 to be configured as a full speed device.

Also testers are encouraged to do some manual tests. Here are some examples:

1. Plug in some real USB devices, suspend system, and then resume. USB devices should still be there.
2. Plug in some real USB devices, suspend system, unplug it, plug in another device, then resume. System should enumerate that new device properly.
3. 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.
4. 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.

32.4.1.8 Platform-Specific API

32.4.1.8.1 BSPCheckConfigPower

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

32.4.1.8.2 BSPUsbSetWakeUp

This function must do what is necessary to enable or disable wakeup on the USB port. For example, 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

32.4.1.8.3 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

32.4.1.8.4 SetPHYPowerMgmt

This function is called by the USB driver when transitioning to/from the suspended state (e.g. during system suspend). The function must do what is necessary to place the transceiver hardware into a suspended (fSuspend == TRUE) or running (fSuspend == FALSE) state.

The standard implementation for <*TGTSOC*>/ADS 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 would need to be modified appropriately.

Parameters:

BOOL fSuspend [in] TRUE: system/controller is going to suspend mode. FALSE: resuming

32.4.2 USB Client Driver

This driver enables the USB device functionality for the i.<TGTPLAT>. It gets activated when a USB Mini B connector is connected to the Mini USB OTG socket. When the i.<TGTPLAT> ADS board is connected to a USB host system (ex: high speed or full speed port of PC), it is enumerated and according to the current

configuration settings, and the appropriate class driver gets loaded on the PC. By default the i.<TGTPLAT> board is configured for USB serial class. i.<TGTPLAT> board can be configured as one of the following USB functions by setting the appropriate environment variable during build (drag/drop from the catalog).

Serial class -- Serial ActiveSync

Mass storage -- expose local storage (ATA hard disk, RAMDISK or other store) as USB drive

RNDIS class (Remote Network Driver Interface Specification)

32.4.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For an overview see section 33.4 and:

Developing a Device Driver –> Windows CE Drivers –> USB Function Drivers –> USB Function Controller Drivers.

User access to the USB client driver is via function drivers such as Mass Storage or RNDIS.

For further details on these USB Function drivers, refer to Windows CE 5.0 Platform Builder help topic:

Developing a Device Driver –> Windows CE Drivers –> USB Function Client Drivers.

Where new function driver code is to be developed, refer to the Function controller driver interface functions (e.g. IssueTransfer) as documented in:

Developing a Device Driver –> Windows CE Drivers –> USB Function Controller Drivers –> USB Function Controller Driver Reference.

32.4.2.2 Client Driver Configuration

The OTG client driver is configured into the BSP build by dragging & dropping the appropriate catalog item. (See 37.1.1) When the “Pure Client” functionality is selected, the OTG port acts only as a device. When “Full OTG functionality” is selected, the OTG port can be either device or host (see transceiver driver configuration).

32.4.2.3 Registry Settings

The USB OTG function/client settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UFN]
```

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization.

Value	Type	Content	Description
Dll	sz	usbfn.dll	Driver dynamic link library

OTGSupport	dword	01	This value must be set to 1 to enable the function/client on the OTG. If no client support is required (host only) then this value can be 0, though the UFN key is not normally configured in the image at all when pure Host function is selected.
OTGGroup	sz	01	This unique string (example “00” to “99”) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance. Only one instance of the OTG is actually supported currently on the *<TGTSOC>* hardware.

32.4.2.4 Device USB Test Modes

The USB 2.0 specification defines PHY-level test modes for USB device ports (see definitions in USB 2.0 specification section 7.1.20). 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 would not be able to leave test mode.

The device port does not by default support the USB test modes. Sample code for test mode support (SET_FEATURE on the device) is included in:

```
PUBLIC\COMMON\OAK\CSP\ARM\FREESCALE\<TGTSOC>\DRIVERS\USBFN\CONTROLLER\MDD
```

In addition, USBFN_TEST_MODE_SUPPORT must be defined during compilation of the CSP USBD device driver library.

32.4.2.5 Unit Test

There is no CETK test case for USB client (function) drivers. The USB Function is tested by configuring the i.<TGTPLAT> ADS board as either USB Serial function or USB Mass storage or RNDIS function and connecting it directly to a Host PC. The test verifies basic USB peripheral/client functionality, including attach, detach, and data transfer.

Separate images must be built and downloaded for each of the three peripheral function tests. Refer 37.4.1.7.2 for building the image.

32.4.2.5.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
Host system	To test if control reaches the Host controller driver.
USB cable having Mini USB OTG plug A at one end and Mini USB OTG plug B on the other side.	For connecting between the host and the device.
ATA drive configured in UDMA mode 2 as DSK1	This is required as a storage device when the board is configured as mass storage class.

32.4.2.5.2 Unit Test Software

The following table shows the software requirements for the USB Function controller driver test.

Requirements	Description
ActiveSync 4.1 and above.	This is the host side software that is required to be available for testing the Serial class functionality.

32.4.2.5.3 Running the USB Function Controller Driver Tests

The following table lists USB Function controller driver tests:

Test Cases	Entry Criteria/Procedure/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 mini USB OTG plug B is connected and the board is turned on and wait till the board boots-up completely. Procedure: 1. Connect the mini USB OTG plug B to the mini 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 will get 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.<TGTPLAT> 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 i.<TGTPLAT> board.
Board configured as USB Mass storage client, with ATA drive as DSK1 mounted, and connected to a host system after the board boots up completely.	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait till the board boots-up completely. Procedure: 1. Connect the mini USB OTG plug B to the mini 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 will get 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 <TGTPLAT> 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 will not be visible from within the <TGTPLAT> system until the system has been reset. The file system should not be used inside the <TGTPLAT> when it is being accessed via 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 mini USB OTG plug B is connected and the board is turned on and wait till 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 etc. 4. On the <TGTPLAT> ADS board, 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 etc. 5. In the Internet explorer on the <TGTPLAT> board, configure the Lan settings as per the local area settings. Expected Result: Browsing the Internet should be possible.

32.4.2.6 Platform-Specific API

32.4.2.6.1 InitializeMux

This function is called to initialize the IOMUX connection within <*TGTSOC*>, 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

32.4.2.6.2 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/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

32.4.3 USB Transceiver Driver (ID Pin Detect Driver -- XCVR)

This driver is responsible for detecting the type of USB connector plugged into the Mini USB OTG socket of i.<*TGTPLAT*>. Upon detection the driver activates the USB host controller driver or USB function controller driver.

32.4.3.1 User Interface

There is no user interface to the transceiver driver. This driver merely manages the USB host or client drivers, which provide the appropriate programming API.

The driver can be configured via its platform-specific routines to provide different behavior for power management (wake-up, D+ soft connect, etc.).

32.4.3.2 Transceiver Driver Configuration

The transceiver driver is configured into the BSP automatically upon dragging and dropping the USB HS OTG catalog item. If transceiver functionality is not required, it can be disabled as described below.

32.4.3.3 Registry Settings

The USB OTG transceiver settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\XVC]
```

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization.

Value	Type	Content	Description
Dll	sz	imx_xvc.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable the transceiver-driven mode switching on the OTG. If no transceiver support is required (host or client only) then this value can be set to 0, though the XVC key will not normally be configured in the image when OTG Pure Host or OTG Pure Client is configured.
OTGGroup	sz	01	This unique string (example “00” to “99”) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance. Only one instance of the OTG is actually supported currently on the *<TGTSOC>* hardware.

32.4.3.4 Unit Test

There is no CETK test case for USB Transceiver driver. The Transceiver driver is tested using the Mini USB OTG plug A and Mini USB OTG plug B. The test is done by manually plugging in the Mini USB OTG plug into the Mini USB OTG socket of i.*<TGTPLAT>* board. The test verifies that the USB host or Function controller driver gets activated on cable plug-in.

32.4.3.4.1 Unit Test Hardware

The following table lists the required hardware to run the unit tests.

Requirements	Description
i.*<TGTPLAT>* board to act as a device.	i.*<TGTPLAT>* board is configured as USB Mass storage class.
USB LS Mouse	To test if control reaches the Host controller driver.
USB cable having A-type plug at one end and Mini USB OTG plug B on the other side. To plug in USB LS mouse, a USB extension cable having mini-A at one end and USB A-type socket at the other end	For connecting between the host and the device.

32.4.3.4.2 Running the Transceiver Test

The following table lists Transceiver tests:

Test Cases	Entry Criteria/Procedure/Expected Results
Idle case when no cable plugged in.	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait till the board boots-up completely. Procedure: When the board is powered and completely booted-up, the board should be idle (and as mass storage client, not verifiable). Expected Result: Device boots up and is stable.
Mass storage client visible from PC	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait till the board boots-up completely. Procedure: When the board is powered and completely booted-up, verify that board responds as a mass storage client when plugged into PC. Expected Result: New storage must be visible on PC.
Mini USB OTG plug-A connected to the mini USB OTG socket of i.<*TGTPLAT*> board and mouse plugged into OTG port via this cable.	Entry Criteria: Unplug board from PC (in previous step). Procedure: 1. Connect the USB HID device (Mouse) which has a Mini USB OTG plug-A to it. The control goes to the USB Host controller driver. 2. The corresponding device gets enumerated and starts functioning. Ex: If a USB mouse is connected, on movement of the mouse, the pointer in the LCD screen is seen moving. Expected Result: Device should start functioning.

32.4.3.5 Platform-Specific API

The transceiver driver library code contains all <*TGTPLAT*> chip-specific implementation, and is located in:

```
PUBLIC\Common\OAK\CSP\ARM\Freescale\<TGTPSOC>\Drivers\USBXVR
```

The transceiver driver operation can be customized through the platform-specific code located in:

```
PLATFORM\<TGTPLAT>\SRC\Drivers\USBXVR
```

The standard implementation located in hwinit.c is provided for the <*TGTPLAT*>/ADS with ISP1504 transceiver attached to the High Speed OTG port.

Customizations would permit different power management and wake-up behavior, including when the device generates soft connect/disconnect (D+ pull-up) or what wake-up conditions are supported when nothing is attached to the OTG port.

The library USB transceiver code communicates with the platform-specific code via callback functions. Only one globally-defined specific routine (RegisterCallback) is required for using this interface.

Standard code is supplied for full transceiver operation using the <*TGTPLAT*>/ADS hardware platform.

32.4.3.5.1 Structure BSP_USB_CALLBACK_FNS

Structure BSP_USB_CALLBACK_FNS is defined in <*TGTSOC*>_usb.h. This is a structure containing all the USB callback functions as called by the USB CSP drivers. Currently only the transceiver driver (USBXVR) uses these callback functions. The callback functions are registered using RegisterCallback() (refer to 2 below).

```
typedef struct {
      // pfnUSBPowerDown - function pointer for platform to call during power down.
      // pfnUSBPowerUp - function pointer for platform to call during power up.
      // Parameter: 1) regs - USB registers
      //            2) pUSBCoreClk - pointer to boolean to indicate the status of USB Core Clk
      //              if it is on or off. Platform is responsible to update this value if they
      change
      //              the status of USBCoreClk. [TRUE - USBCoreClk ON,  FALSE - USBCoreClk OFF]
      //             3) pPanicMode - pointer to boolean to indicate the status of panic mode
      //              if it is on or off. Platform is responsible to update this value if they
      change
      //            the status of panic mode. [TRUE - PanicMode ON,  FALSE - USBCoreClk OFF]
      void (*pfnUSBPowerDown)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk, BOOL *pPanicMode);
      void (*pfnUSBPowerUp)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk, BOOL *pPanicMode);
      // pfnUSBSetPhyPowerMode - function pointer for platform to call when they want to
      suspend/resume the PHY
      // Parameter: 1) regs - USB registers
      //            2) bResume - TRUE - request to resume, FALSE - request suspend
      void (*pfnUSBSetPhyPowerMode)(CSP_USB_REGS *regs, BOOL bResume);
      } BSP_USB_CALLBACK_FNS;
```

32.4.3.5.2 pfnUSBPowerDown

This callback function is called during the WinCE power down sequence. The actual platform specific power down routine should be registered as this callback function. Note: this function is only called if the system is in USB transceiver mode only (i.e. when nothing is attached to the OTG port.).

There is no standard implementation for this callback, since by default the transceiver driver will automatically suspend the port when nothing is attached. This enables wake-up on device or host attachment, and enables the D+ pull-up during the suspended condition.

Parameters

*CSP_USB_REGS *regs*

[in] Mapped pointer to the USB registers in <*TGTSOC*>, from physical address space to a non-paged, process-dependent address space. This is mapped during the transceiver initialization routine (XVC_Init).

*BOOL *pUSBCoreClock*

[in/out] Pointer to a Boolean variable in transceiver to indicate whether the USB Core Clock has been stopped.

The platform-specific callback function must update this flag to reflect the current USB Core Clock status, if the status of the USB Core Clock is changed within the platform code (for example using DDKClockSetGatingMode()). This ensures consistency of the clock status within the CSP transceiver driver.

TRUE – USB Core Clock is running

FALSE – USB Core Clock is stopped

*BOOL *pPanicMode*

[in/out] Pointer to a Boolean variable to indicate whether the USB has requested for system voltage to remain in Panic Mode or not. The callback function must update this flag to reflect the current Panic Mode status, if this status is changed within the platform code (for example using DDKClockEnablePanicMode(). This ensures consistency of the Panic Mode status within the CSP transceiver driver.

TRUE – Panic mode is currently requested for the USB

FALSE – Panic mode is not currently requested for the USB

32.4.3.6 pfnUSBPowerUp

Similar to pfnUSBPowerDown, this is called during the WinCE power up sequence. The actual platform specific power up (resume) routine should be registered to this pointer. This is only called when USB is in transceiver mode (i.e. when nothing is attached to the OTG port).

There is no standard implementation for this callback, since by default the transceiver driver will automatically suspend the port when nothing is attached and the port need not perform any wake-up activity until a device or host attachment is detected.

Parameters For parameter details see pfnUSBPowerDown, section 37.4.3.5.2.

32.4.3.6.1 pfnUSBSetPhyPowerMode

This function is called when the system is in USB transceiver mode, with no USB activity. With standard implementation on <*TGTPLAT*>/ADS, if the system is in transceiver mode and there is no activity in USB port for 1 second, the transceiver driver will suspend the ULPI PHY (in this case, it is ISP1504, disable the USB Clock gating, and set the system to non-panic mode (allowing core voltage to drop).

When there is USB activity (e.g. device attach), the transceiver driver sets the system to panic mode (requiring core voltage to stay high using DDKClockEnablePanicMode(), Supported for MX31), enables USB Clock gating and puts the ULPI PHY transceiver to resume.

This callback function is responsible for handling the suspend and resume of ULPI PHY transceiver. The developer must register this pointer with the actual platform specific function for suspend and resume of ULPI PHY transceiver. Custom wake-up conditions can be enabled here.

Parameters

*CSP_USB_REGS *regs*

[in] Mapped pointer to the USB registers in <*TGTPLAT*>, from physical address space to a non-paged, process-dependent address space. This is mapped during the transceiver initialization routine (XVC_Init).

BOOL resume

[in] This boolean variable indicates whether the callback function must resume or suspend the ULPI PHY transceiver.

TRUE – callback function must resume transceiver activity
FALSE – callback function must suspend transceiver activity

32.4.3.6.2 RegisterCallback

This is used to register all the callback functions defined in BSP_USB_CALLBACK_FNS. This function is called by the USB driver during the initialization process of the transceiver driver (XVC_Init). The developer must implement a function by this name in their platform directory,

A standard implementation is provided for the ISP1504/ISP1301 transceiver of the <*TGTPLAT*>/ADS.

When no callback function is required, those elements of the BSP_USB_CALLBACK_FNS structure should be initialized to NULL.

Parameters

*BSP_USB_CALLBACK_FNS *pFn*

[in/out] Pointer to BSP_USB_CALLBACK_FNS structure for the developer to register the callback function inside the BSP_USB_CALLBACK_FNS. The callback functions inside this structure will be used by the CSP transceiver code.

32.4.4 Power Management

There are four aspects of power management for the USB device drivers:

1. Special <*TGTPLAT*> Vcore requirements
2. Clock gating to the USB peripheral block within the <*TGTPLAT*>
3. Setting the transceiver to a lower power mode or suspend
4. Transceiver auto-power-down on inactivity

The USB device driver(s) support an ON and OFF/standby (low power) state, with wake-up capability. The ON state is entered whenever a host or device is attached to the relevant USB port. The driver enters the standby state is automatically after timeout with no device or host attached to the USB port. As well, the standby state is entered when the system suspends. (In the latter case, system wake-up capability is enabled for the port).

32.4.4.1 Special MX31 Vcore requirements

When ULPI-bus transceivers are used with the USB controller (e.g. ISP1504 transceivers for High Speed OTG port and High Speed Host 2 port on MX31/ADS), normal DVFS scaling of the MX31 Vcore must be suspended whenever there is potential of ULPI bus communication. This is the case whenever a device is connected (in host mode) or the device is connected to a host (in client mode). The USB OTG Transceiver driver, and USB Host and Client drivers constrain the DVFS behavior by calling DDKClockEnablePanicMode() whenever a device or host connection is detected, and calling DDKClockDisablePanicMode() when a timeout period expires with no device or host connected to the port. There is no user configuration required here; only the effect on DVFS (DVFC driver) behavior need be noted.

Note : The boards with MX31 TO2 onwards do not have this requirement.

32.4.4.2 Clock Gating

The USB driver(s) for the various USB ports automatically manage clock gating to the <*TGTPLAT*> USB controller cores. The drivers for the ports coordinate their use of the USB core clock, and when nothing is connected on any of the ports (all drivers are in their lowest power state) the clock is gated on or off using:

```
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_USBOTG, DDK_CLOCK_GATE_MODE_ENABLED_ALL)
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_USBOTG, DDK_CLOCK_GATE_MODE_DISABLED)
```

32.4.4.3 Transceiver Auto Power Down

The USB transceivers automatically enter a lower-power/suspended mode when no USB traffic is detected for several milliseconds. This internally sets a suspended state for the USB port. Software timeout is used to establish whether the driver power mode can be switched to its lowest power state (see below).

32.4.4.4 Transceiver Power Mode

Software timeout is used to establish whether the transceivers and their related bus (e.g. ULPI-bus for ISP1504 connection to <*TGTPLAT*>) needs to be set to a suspended condition. In the lowest-power state, the transceiver is configured to generate wake-up signalling on attachment of devices or host (to the OTG port). The transceiver driver provides callback routines to manage this transition.

32.4.4.5 PowerUp

Each of the OTG client, host and transceiver drivers have PowerUp routine associated. (For the host driver, this is referenced via the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- verify the wake-up conditions via the BSPUsbCheckWakeUp() platform routine
- stop the host controller
- suspend the relevant port
- set the PHY to low power mode using SetPHYPowerMgmt(TRUE) platform routine
- disable panic mode for the core voltage (DDKClockDisablePanicMode())
- gate the USB peripheral block clock

For the client, the routine does the following:

- ungate the USB peripheral block clock
- enable panic mode for the core voltage (DDKClockEnablePanicMode())
- force the port to resume
- disable the wake-up conditions
- enable the interrupts and start the USB controller

For the transceiver driver, the PowerUp routine calls the relevant platform-specific callback routine, pfnUSBPowerUp().

Under normal circumstances there is nothing to be done in this routine, since the OTG port is normally in a suspended state within the transceiver mode. (It is only in transceiver mode when nothing is connected to the port, and thus has already been automatically suspended).

32.4.4.6 PowerDown

As for the PowerUp routine, OTG client, host and transceiver drivers have PowerUp routine associated. (For the host driver, this is referenced via the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- verify the wake-up conditions via the BSPUsbCheckWakeUp() platform routine
- stop the host controller
- suspend the relevant port
- set the PHY to low power mode using SetPHYPowerMgmt(TRUE) platform routine
- disable panic mode for the core voltage (DDKClockDisablePanicMode())
- gate the USB peripheral block clock

For the client, the routine does the following:

- stop the USB controller
- clear any outstanding interrupts
- enable appropriate wake-up condition
- suspend the relevant port (suspends the PHY)
- disable core voltage panic mode (DDKClockDisablePanicMode())
- gate the USB peripheral block clock

For the transceiver driver, the PowerDown routine calls the relevant platform-specific callback routine, pfnUSBPowerDown().

Under normal circumstances there is nothing to be done in this routine, since the transceiver remains in its suspended state while nothing is connected to the port. Should any attachment have been made, the transceiver would wake through its wake-up mechanism and launch the appropriate (client or host) driver.

32.4.4.7 Wake-up Conditions

USB host ports (or the OTG port in transceiver mode) will generate a system wake-up condition when a device is attached.

The USB device port (the OTG port in transceiver mode) will generate a system wake-up condition when attached to a host.

No wake-up is generated when a device is detached, or the host is detached. This behavior can be customized in the platform code, as well as the current state of D+ (soft attach) for example.

Note that if a device or host is attached before the system is suspended, then any reconfiguration of the port attachment will not wake the system (e.g. detachment of the host and subsequent attachment of a device).

When the system resumes (either due to USB wake-up or other wake-up cause) then the USB mode is re-evaluated. If devices are attached then they are re-enumerated and their class drivers re-loaded. If a host is attached to the OTG port, then the host will detect the OTG as client and attachment and launching of the function driver will take place.

32.4.5 Function Drivers

The function drivers can be configured into the image via the Windows CE 5.0 Platform Builder catalog, and are located at:

Device Drivers -> USB Function –> USB Function Clients

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 will apply):

```
[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"
```

Unless the BSP is configured with persistent registry storage, it will only make sense to configure a single function driver into the image, and this one will become default.

Note: when no USB client functionality is included in the image (No OTG port, or OTG Pure Host only), then ensure that no function drivers have been configured. If function drivers are configured, then USB client driver libraries will also be included in the image via logic in:

```
PUBLIC\CEBASE\OAK\Misc\winceos.bat
```

32.4.5.1 Mass Storage Function

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	\PLATFORM\<TGTPLAT>\SRC\DRIVERS\USBMSFN
Import Library	USBMSFN_LIB.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. This device used by default for this local data store is “DSK1”, and could be configured as ATA drive or RAMDISK or even a USB drive attached to a different USB host port, depending on the BSP configuration.

The name DSK1 is associated with the Mass Storage function via the value “DeviceName” under the key:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Mass_Storage_Class]
```

which is imported by default when SYSGEN_USBFN_STORAGE is defined, from:

```
PUBLIC\Common\OAK\Files\common.reg
```

This default registry entry can (should) be copied into platform.reg and modified to over-ride the defaults. This will allow 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
```

32.4.5.2 Serial Function

The primary use for Serial function is ActiveSync.

Driver Attribute	Definition
CSP Driver Path	N/A
PUBLIC driver path	PUBLIC\Common\OAK\Drivers\USBFN\CLASS\SERIAL
CSP Static Library	N/A
Platform Driver Path	N/A
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 connection to PC with ActiveSync version 4.1 installed. Please see microsoft.com to download the latest ActiveSync software for the PC.

When SYSGEN_USBFN_SERIAL is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

This registry entry can (should) be copied into platform.reg and modified to over-ride the defaults. This will allow 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
```

32.4.5.3 RNDIS Function

The RNDIS function allows communication over USB to be supplied to ethernet NDIS interface of protocol stack.

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	N/A
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

Note: RNDIS function has been tested using PC RNDIS class driver as located at:

```
PUBLIC\Common\OAK\Drivers\ETHDBG\Rndismini\HOST\usb8023.inf
%WINDIR%\System32\drivers\usb8023.sys
```

When SYSGEN_USBFN_ETHERNET is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

This registry entry can (should) be copied into platform.reg and modified to over-ride the defaults. This will allow 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
```

32.4.6 Class Drivers

All host ports (OTG Host, High Speed Host (H2), and Full Speed Host (H1)) support the same class drivers, and this configuration is common to all host ports.

For details of supported class drivers and their configuration, see section 37.5.2.

32.5 USB High Speed Host (H2) Driver

The USB High Speed Host (H2) driver provides USB2.0 host and device support for the USB High Speed host port of the <*TGTPLAT*>.

The USB host driver has class support for mass storage, HID, printer, and RNDIS clients

32.5.1 USB HS Host 2 (H2) Driver Summary

32.5.1.1 Host Driver Summary

32.5.1.1.1 For MX31/MX32

Driver Attribute	Definition
Target Platform (TGTPLT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	..\CSP\ARM\Freescale\<*TGTSOC*>\Drivers\USBH
CSP Static Library	<TGTSOC>_ehci_lib.lib <TGTSOC>_Ehcdmdd.lib <TGTSOC>_hcd2lib.lib
Platform Driver Path	\PLATFORM\<*TGTPLAT*>\SRC\Drivers\USBH\HSH2
Import Library	N/A
Driver DLL	hcd_hsh2.dll
Catalog Item	Third Party –> BSPs –> Freescale <*TGTPLAT*>: ARMV4I –> Device Drivers –> USB Devices –> USB High Speed Host 2
SYSGEN Dependency	SYSGEN_USB
BSP Environment Variable	BSP_USB BSP_USB_HSH2

32.5.2 Class Drivers

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 to be completed on any class driver.

These are standard Microsoft-supplied drivers, and are available by drag & drop from the catalog:

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 33.5.2.1 below)
Keyboard	SYSGEN_USB_HID _KEYBOARD	.. −> Keyboard HID Device (and see additional configuration in section)
	SYSGEN_USB_HID _MOUSE	.. −> Mouse HID Device (and see additional configuration in section 33.5.2.2 below)
RNDIS	SYSGEN_ETH_USB_HOST	.. −> USB Remote NDIS Class Driver
Storage	SYSGEN_USB_STORAGE	.. −> USB (mass) Storage Class Driver

Drag and drop all the class drivers required for the USB Host class.

Note: When no USB host ports are configured in the image, ensure that no class drivers are selected, otherwise host libraries will be included by default from logic in:

```
PUBLIC\CEBASE\OAK\Misc\winceos.bat
```

32.5.2.1 Printer

For printer support, you will also require printer driver/protocol support. For example, you may need to include PCL:

Catalog Item	Configuration Flag	Catalog Item
PCL	SYSGEN_PCL	Device Drivers −> Printer Devices −> PCL Printer Driver

For more information, see Windows CE Platform Builder help topic:

Developing a Device Driver −> Windows CE Drivers −> USB Host Drivers −> USB Host Client Drivers −> USB Host Printer Client Driver

32.5.2.2 HID Mouse

For mouse support, you will also require the cursor in order to test/use the mouse.

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS −> Shell and User Interface −> User Interface −> Customizable UI −> Mouse

32.5.2.3 HID Keyboard

The HID keyboard can be used simultaneously with <*TGTPLAT*>/ADS Keyboard.

32.5.3 Requirements

1. The High Speed H2 driver must support USB specification 2.0.

2. The H2 driver shall support mass storage, printer, HID and RNDIS classes
3. When nothing is attached to the H2 port, the driver shall automatically configure the USB controller and transceiver into a low power state
4. When the system is suspended with nothing attached to the H2 port, the system shall wake upon device attachment to the port.
5. When the system is suspended while H2 port is connected to a device, the system shall remain suspended when the device is detached.
6. When the system resumes after suspend, any attached devices shall be enumerated and their class drivers loaded appropriately.

32.5.4 Hardware Operation

The USB H2 port on the ADS is connected via a high-speed ULPI transceiver to an EHCI controller within the <*TGTSOC*>. The <*TGTSOC*> controller contains speed translation logic, to allow direct connection of Full Speed and Low Speed peripherals (as well as High Speed peripherals) to the controller.

32.5.4.1 Conflicts with other Peripherals and Catalog Options

32.5.4.1.1 MX31/MX32 Interface Conflicts

Because of IOMUX and pin conflicts, H2 can not currently be configured with ATA, UART5 and PCMCIA devices. (Also see section 37.7.1)

Currently limitations in the support for the transaction (speed) translation layer mean that only high speed hub is supported, and full and low-speed devices must be directly attached to the port.

32.5.5 Software Operation

32.5.5.1 High Speed USB Host 2 Configuration

The USB H2 driver is configured into the BSP build by dragging & dropping the appropriate catalog item. Additional classes to be supported must also be selected from the Core OS catalog.

The internal <*TGTSOC*> USB H2 signals can be multiplexed to a choice of pins on IC, as described for the IOMUX in the hardware reference manual. The ADS also supports multiplexing these signals through a choice of buffers in one of two configurations.

The USB H2 driver currently supports only one specific hardware configuration, which defines the other on-chip peripherals and devices with which the H2 device can be simultaneously configured in the BSP. (As per section 37.3.1)

32.5.5.2 Memory Configuration

The physical memory pool used by the High Speed Host driver is configured by registry key PhysicalPageSize as defined in section 37.5.5.3 below. For discussion of the pool see section 37.4.1.3.

32.5.5.3 Registry Settings

The USB H2 settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSH2]
```

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization.

Default values are contained in hsh2.reg.

Value	Type	Content	Description
Dll	sz	hcd_hsh2.dll	Driver dynamic link library
OTGSupport	dword	00	This controller is not OTG capable
OTGGroup	sz	02	This instance grouping shall be unique from the one used by the USB OTG port, if it is included in the build.
HcdCapability	dword	4	HCD_SUSPEND_ON_REQUEST. Note: HCD_SUSPEND_RESUME is always assumed.
PhysicalPageSize	dword	20000	This value represents the number of bytes allocated for the physical memory pool of the H2 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 will be at least 3x as large as the largest data buffer transferred using IssueTransfer().

32.5.5.4 USB Test Mode

The USB 2.0 specification defines PHY-level test modes for USB host ports (see definitions in USB 2.0 specification section 7.1.20).

The High Speed Host driver support “Packet” test mode, which is configured by compiling the BSP with the compilation flag H2_TEST_MODE defined within bsp_cfg.h:

```
#define H2_TEST_MODE
```

This configures the appropriate host controller within the platform-specific hardware initialization function (ConfigH2()), located in:

```
PLATFORM\MX31\SRC\Drivers\USBH\Common\hwinit.c
PLATFORM\MX32\SRC\Drivers\USBH\Common\hwinit.c
```

The test mode is entered upon initialization, and can not be exited. Normal USB operation is disabled when test mode support is compiled into the image.

32.5.5.5 Unit Test

Test of H2 High Speed host port is similar to the testing of the OTG High Speed Host (see section 37.4.1.7). Device classes for Printer, RNDIS, HID and Mass Storage must be supported.

Examples of each device, in High, Full, and Low speed (mouse, keyboard), are attached and detached, and data transfer verified.

In addition, the Windows CE 5.0 USB Host CETK is executed as described in 37.4.1.7.1.

32.5.5.6 Power Management

Power management of the High Speed Host port is the same as for Host mode within the OTG High Speed driver (see section 37.4.4).

The standard implementation supports the ISP1504 transceiver as for the High Speed OTG port.

32.5.5.7 API

User access and customization of the High Speed host driver are the same as described in 37.4.1.8.

32.6 USB Full Speed Host (H1) Driver

The USB H1 driver provides support for Full Speed and Low Speed peripherals, attached to the H1 port of the <*TGTPLAT*>. The <*TGTPLAT*> contains an on-chip full speed host controller, and the <*TGTPLAT*>/ADS has this port connected to an ISP1301 Full Speed transceiver.

The USB host driver can be configured with class support for mass storage, HID, RNDIS, and printer clients.

32.6.1 USB Full Speed Host (H1) Driver Summary

32.6.1.1 For MX31/MX32

Driver Attribute	Definition
Target Platform (TGTPLT)	MX31, MX32ADS
Target SOC (TGTSOC)	MX31, MX32
CSP Driver Path	PUBLIC*\CSP\ARM\Freescale\<*TGTSOC*>\Drivers\USBD
CSP Static Library	<*TGTSOC*>_ehci_lib.lib <*TGTSOC*>_Ehcdmdd.lib <*TGTSOC*>_hcd2lib.lib
Platform Driver Path	\PLATFORM\<*TGTPLAT*>\SRC\DRIVERS\USBH\FSH1
Import Library	N/A
Driver DLL	hcd_fsh1.dll
Catalog Item	Third Party −>BSPs −>Freescale <*TGTPLAT*>: ARMV4I −> Device Drivers −> USB Devices −> USB Full Speed Host 1
SYSGEN Dependency	SYSGEN_USB
BSP Environment Variable	BSP_USB BSP_USB_FSH1

The host driver requires one or more class drivers to be loaded. These may each also require additional components (such as SYSGEN_HID_MOUSE requires SYSGEN_CURSOR). See 37.1.2 for details.

32.6.2 Requirements

1. The USB Full Speed (H1) driver shall support the Full Speed signalling and protocol of USB specification 2.0.
2. The H1 driver shall support a serial Full Speed transceiver (as implemented on the <*TGTPSOC*>/ADS).
3. The H1 driver shall support mass storage, printer, HID and RNDIS classes, supporting attached Low Speed, Full Speed and High Speed (operating in Full Speed mode only) devices.
4. When nothing is attached to the H1 port, the driver shall automatically configure the USB controller and transceiver into a low power state
5. When the system is suspended with nothing attached to the H1 port, the system shall wake upon device attachment to the port.
6. When the system is suspended while H1 port is connected to a device, the system shall remain suspended when the device is detached.
7. When the system resumes after suspend, any attached devices shall be enumerated and their class drivers loaded appropriately.

32.6.3 Hardware Operation

The USB H1 Full Speed port on the ADS is connected via a full speed serial transceiver (ISP1301) to the host 1 controller within the <*TGTSOC*>.

The H1 port can be used in combination with the High Speed OTG and High Speed Host (H2) ports.

32.6.3.1 Conflicts with other Peripherals and Catalog Options

32.6.3.1.1 MX31/MX32 Interface Conflicts

Due to pin conflicts, the H1 port can not be configured within a BSP together with CSPI ,ATA and UART3 devices. (See also section 37.7.1)

An MX31/MX32 USB controller limitation also means that Low Speed devices can not be attached to a High Speed hub which is attached to the Full Speed H1 port.

32.6.4 Software Operation

32.6.4.1 Full Speed USB Configuration

The USB H1 driver is configured into the BSP build by dragging & dropping the appropriate catalog item. Additional classes to be supported must also be selected from the Core OS catalog.

The internal <*TGTSOC*> USB H1 signals can be multiplexed to a choice of pins on IC, as described for the IOMUX in the hardware reference manual. The ADS also supports multiplexing these signals through a choice of buffers in one of two configurations. The USB H1 driver currently supports only one specific hardware configuration, which defines the other on-chip peripherals and devices with which the H1 device can be simultaneously configured in the BSP.

32.6.4.2 Memory Configuration

The physical memory pool used by the Full Speed Host driver is configured by registry key PhysicalPageSize as defined in section 37.6.4.3 below. For discussion of the pool see section 37.4.1.3.

32.6.4.3 Registry Settings

The USB H1 settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_FSH1]
```

The values under that registry key should be defined in platform.reg. These can be qualified with the BSP_USB_FSH1 system variable for configurable catalog item support. Default values are contained in fsh1.reg.

Value	Type	Content	Description
Dll	sz	hcd_fsh1.dll	Driver dynamic link library
OTGSupport	dword	00	This controller is not OTG capable
OTGGroup	sz	03	This instance grouping shall be unique from the one used by the USB OTG port, if it is included in the build.
PhysicalPageSize	dword	20000	This value represents the number of bytes allocated for the physical memory pool of the H1 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 will be at least 3x as large as the largest data buffer transferred using IssueTransfer().
HcdCapability	dword	4	HCD_SUSPEND_ON_REQUEST Note: HCD_SUSPEND_RESUME is always assumed.

32.6.4.4 USB Test Mode

No USB PHY test mode support is configured for the Full Speed host port.

32.6.4.5 Unit Test

Test of H1 Full Speed host port is similar to the testing of the other USB Host ports, with simpler device speed combinations. Device classes for Printer, RNDIS, HID and Mass Storage must be supported.

Examples of each device, in Full and Low speed (mouse, keyboard), are attached and detached, and data transfer verified.

The Windows CE USB Host CETK kit is also executed on the Full Speed port, as described in 37.4.1.7.1

32.6.4.6 Power Management

Power management of the Full Speed port is the same as for Host mode within the OTG High Speed driver (see section 33.4.4).

The standard implementation supports the ISP1301 transceiver with serial interface, and in this case no special control of the transceiver-<*TGTSOC*> bus is required during suspend/resume. In other senses, all operation is the same.

32.6.4.7 API

User access and customization of the Full Speed host driver are the same as described in 37.4.1.8.

Normal user access is via class drivers.

32.7 Basic Elements for Driver Development

This chapter provides details of the basic elements for driver development in the <*TGTPLAT*> BSP.

32.7.1 BSP Environment Variables

Table 32-1. <TGTPLAT> BSP Environment Variables Summary

Names	Definition
BSP_USB	Set to configure USB in BSP
BSP_USB_HSOTG_XVC	Set to enable Full OTG functionality (transceiver 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_FSOTG_CLIENT	Set to include USB client functionality on Full Speed OTG port.
BSP_USB_HSOTG_HOST	Set to include USB host functionality on High Speed OTG port.
BSP_USB_FSOTG_HOST	Set to include USB host functionality on Full Speed OTG port.
BSP_USB_FSH1	Set to enable USB Full Speed Host port driver
BSP_USB_HSH2	Set to enable USB High Speed Host port driver

Pin conflicts between default driver implementations for the MX31/MX32 pin muxing (platform-specific implementation) mean certain configurations are mutually exclusive, as listed in the following table.

Table 32-2. Mutual Exclusive Driver Summary

Functionality yes = Required no = Not permitted — = Don't care	BSP_ATA	BSP_CSPIBUS	BSP_USB	BSP_USB_HSOTG_XVC	BSP_USB_HSOTG_CLIENT	BSP_USB_HSOTG_HOST	BSP_USB_FSH1	BSP_USB_HSH2
ATA disk drive	yes	no					no	no
High Speed OTG Port full function (Host + Client)			yes	yes	yes	yes		
High Speed OTG Port Pure Host only			yes			yes		
High Speed OTG Port Pure Client only			yes		yes			
Full Speed Host (H1)	no	no					yes	
High Speed Host (H2)	no	no						yes

32.7.2 Dependencies of Drivers

The following table summarizes the Microsoft defined environment variables used in the BSP.

Table 32-3. . USB Driver

Names	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

Chapter 33
Video Processing Unit (VPU)

The Video Processing Unit (VPU) is a multi-media video processing module on i.MX32. The i.MX32 supports decoding for H.263, MPEG4, H.264 and VC-1 standards up to D1 resolution. This chapter provides the following contents:

- Describe the brief information of VPU DLL.
- Describe the API provided by Freescale which allows complete access to the full functionality of the VPU.
- Describe the VPU control scheme based on the API with some practical programming issues.

This document is intended for application developers who use the VPU to implement high performace video codec and need to understand and gain access to the functionality provided by the VPU.

33.1 VPU driver summary

The following table provides a summary of source code location, library dependencies and other BSP information:

Driver Attribute	Definition
Target Platform (TGTPLT)	MX32ADS
Target SOC (TGTSOC)	MX32
CSP Driver Path on WINCE 5.0 SOC Driver Path on WINCE 6.0	..\CSP\ARM\FREESCALE\MX32\DRIVERS\VPU ..\SOC\FREESCALE\MX32_FSL_V1\VPU
CSP Static Library on WINCE 5.0 SOC Static Library on WINCE 6.0	mx32_VPU.lib vpu_mx32_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT> \SRC\DRIVERS\VPU
Import Library	N/A
Driver DLL	vpu.dll
Catalog Item	Third Party −>BSPs −>Freescale <TGTPLAT> −>Device Drivers −>Video Processing Unit
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_VPU=1

33.2 Requirements

The VPU driver should meet the following requirements:

1. The driver shall support all APIs defined by Freescale.

2. The driver shall support the interrupt mode.
3. The driver shall support the multi-task function provided by the hardware.

33.3 Hardware Operation

Refer to the chapter on Video Codec (Video_Codec) in the hardware specification document for detailed hardware operation and programming information.

33.3.1 Conflicts with other SoC peripherals

No conflicts.

33.4 Software Operation

33.4.1 Communicating with the VPU

The VPU DLL requires the special initialization required by interrupt processing, which initializes the VPU hardware interrupt with the kernel and creates the interrupt service thread (IST). The special intialization required by the VPU is performed when the DLL is loaded by the Device Manager, because the VPU DLL is configured as Build-In driver. But the VPU DLL also can be loaded into the respective process space when applications want to utilize the APIs exported by the VPU, which simply need to link to the VPU export library and invoke the exported APIs.

33.4.2 Power Management

Power management is not implemented in this version.

33.4.3 Codecs Registry Settings

The following registry keys are required to properly load the decoder drivers:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\VPU]
   "Dll"="vpu.dll"
   "Order"=dword:5
```

33.5 Unit Test

The VPU can be tested using a custom VPU test application.

33.5.1 Unit Test Hardware

33.5.1.1 Unit Test Hardware for MX32

The following table lists the required hardware to run VPU application test on MX32ADS.

Requirements	Description
Sharp LQ035Q7DB02 (QVGA) or Epson LSFxxxxxT00(QVGA)	Display panel required for display of graphics data.

33.5.2 Unit Test Software

33.5.2.1 Unit Test Software for MX32

Requirements	Description
vpu.dll	The VPU DLL to export the API.
dectest.exe	Decoding the bitstream date file and displaying the decoded images on the LCD

33.5.3 Running the VPU Application Test

33.5.3.1 Running the VPU Application Test on MX32

Test Cases	Entry Criteria/Procedure/Expected Results
Decoding Test	Entry Criteria: The WinCE OS image with LCD support, CE Target Control & KITL support, bitstream data file. Procedure: 1. Change the vpu.cfg configuration file according to the bitsteam format, Image size you want to display, frame rate and other parameters. You can get detailed inforamtion from readme.txt and vpu.cfg files. 2. Run the CE Target Control Debugging command " ***s dectest.exe [path]\vpu.cfg***". Expected Result: The decoded image is displayed on the VGA LCD panel.

33.6 VPU Driver API Reference

The API functions are defined by Freescale and a third party IP vender. For details, please refer to the appropriate docement \WINCE500\SUPPORT\MX32\APPS\VPUAPP\vpu_api_user_manual_mx32.pdf or \WINCE600\SUPPORT\MX32\APPS\VPUAPP\vpu_api_user_manual_mx32.pdf.

33.7 Sample Application

This section describes how to build and run the custom VPU test application. The VPU application, named DECTEST, can be found in one of the following locations:
\WINCE500\SUPPORT\MX32\APPS\VPUAPP or \WINCE600\SUPPORT\MX32\APPS\VPUAPP

This application provides an example of how to implement a video decoder using the VPU video acceleration hardware.

33.7.1 System Requirements on MX32

In order to build and the VPU application, the following requirements must be met:

- The OS image must be built with the VPU driver from the Catalog.
- There must be access to raw bitstream data files (H.263, MPEG4, H.264 and VC-1) for decoding test (Note: you might need to include SD Host Controller drivers or storage drivers, such as ATA, Nand, SD from the Catalog to enable fast loading of test data.)

33.7.2 Building the WinCE Image and VPU Test Application

33.7.2.1 Building the WinCE Image

1. Include **Third Party->BSPs->Freescale MX32 ADS:ARMV4I->Device Drivers->Video Processing Unit** on Windows CE 5.0 or **Third Party->BSP->Freescale i.MX32 ADS:ARMV4I->Device Drivers->VPU** on Windows CE 6.0.
2. Optionally include **Third Party->BSPs->Freescale MX32 ADS:ARMV4I->Device Drivers->SD Host Controller (** or **Storage Drivers)** on Windows CE 5.0 or **Third Party->BSP->Freescale i.MX32 ADS:ARMV4I->Device Drivers->SD Host Controller (** or **Storage Drivers)** on Windows CE 6.0.

33.7.2.2 Building and Running the VPU Test Application

1. Click **Build OS** -> **Open Release Directory** on Windows CE 5.0 or **Build** -> **Open Release Directory in Build Window** on Windows CE 6.0 to open the command prompt.
2. Run command "*set wincerel=1*" in command prompt window.
3. Change the current path to \WINCE500\SUPPORT\MX32\APPS\VPUAPP or \WINCE600\SUPPORT\MX32\APPS\VPUAPP.
4. Build the application with "*build -c*" command.
5. Run the VPU application from the CE Target Control with the command "*s dectest \release\vpu.cfg*" (if vpu.cfg file is copied to \release directory, for example. Make sure your parameters set in vpu.cfg file is correct for your bitstream and your h/w display. For detailed information, refer to the readme.txt and vpu.cfg files in WINCE500\SUPPORT\MX32\APPS\VPUAPP or WINCE600\SUPPORT\MX32\APPS\VPUAPP).